21世纪高等学校规划教材 | 计算机应用



Android系统结构及应用编程

佘堃　段弘　唐雪飞　编著

清华大学出版社

21世纪高等学校规划教材·计算机应用

Android 系统结构及应用编程

佘 堃 段 弘 唐雪飞 编著

清华大学出版社
北 京

内 容 简 介

本书主要介绍了目前最流行的移动操作系统 Android 系统结构和编程基础，内容涵盖了大部分最常用和最实用的开发常识和技巧。全书以对 Android 的背景常识、发展历史的介绍作为入口点，进一步介绍了 Android 系统结构、编程结构以及应用编程基础，简单介绍了用户界面(UI)开发、数据存储与共享，深入探讨了多进程与多线程开发、多媒体编程、网络开发、Android WebKit 开发、NDK 入门、游戏案例以及 Chrome 扩展等。

本书编写的原则是叙述方式通俗易懂，但在内容上又不失水准；特点是覆盖全面、详尽，搭配完整的源代码及注释，再加上图文结合的形式可以使读者在学习的过程中更加得心应手。本书可作为 Android 中高级开发人员的开发手册，是帮助 Android 初级开发人员的进一步进阶。

图书在版编目(CIP)数据

Android 系统结构及应用编程/佘堃，段弘，唐雪飞编著. —北京：清华大学出版社，2013.1
ISBN 978-7-302-30208-7

Ⅰ. ①A…　Ⅱ. ①佘…　②段…　③唐…　Ⅲ. ①移动终端—应用程序—程序设计—教材
Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2012)第 228442 号

责任编辑： 魏江江　李　晔
封面设计： 傅瑞学
责任校对： 白　蕾
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载： http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 23.75　　**字　　数：** 577 千字
版　　次： 2013 年 1 月第 1 版　　**印　　次：** 2013 年 1 月第 1 次印刷
印　　数： 1～　000
定　　价： 　.00 元

产品编号：046140-01

编审委员会成员

（按地区排序）

出版说明

随着我国改革开放的进一步深化，高等教育也得到了快速发展，各地高校紧密结合地方经济建设发展需要，科学运用市场调节机制，加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度，通过教育改革合理调整和配置了教育资源，优化了传统学科专业，积极为地方经济建设输送人才，为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是，高等教育质量还需要进一步提高以适应经济社会发展的需要，不少高校的专业设置和结构不尽合理，教师队伍整体素质亟待提高，人才培养模式、教学内容和方法需要进一步转变，学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007 年 1 月，教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》，计划实施“高等学校本科教学质量与教学改革工程”(简称“质量工程”)，通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容，进一步深化高等学校教学改革，提高人才培养的能力和水平，更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部“质量工程”的过程中，各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势，对其特色专业及特色课程(群)加以规划、整理和总结，更新教学内容、改革课程体系，建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上，经教育部相关教学指导委员会专家的指导和建议，清华大学出版社在多个领域精选各高校的特色课程，分别规划出版系列教材，以配合“质量工程”的实施，满足各高校教学质量和教学改革的需要。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作，提高教学质量的若干意见》精神，紧密配合教育部已经启动的“高等学校教学质量与教学改革工程精品课程建设工作”，在有关专家、教授的倡议和有关部门的大力支持下，我们组织并成立了“清华大学出版社教材编审委员会”(以下简称“编委会”)，旨在配合教育部制定精品课程教材的出版规划，讨论并实施精品课程教材的编写与出版工作。“编委会”成员皆来自全国各类高等学校教学与科研第一线的骨干教师，其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求，“编委会”一致认为，精品课程的建设工作从开始就要坚持高标准、严要求，处于一个比较高的起点上。精品课程教材应该能够反映各高校教学改革与课程建设的需要，要有特色风格、有创新性(新体系、新内容、新手段、新思路，教材的内容体系有较高的科学创新、技术创新和理念创新的含量)、先进性(对原有的学科体系有实质性的改革和发展，顺应并符合 21 世纪教学发展的规律，代表并引领课程发展的趋势和方向)、示范性(教材所体现的课程体系具有较广泛的辐射性和示范性)和一定的前瞻性。教材由个人申报或各校推荐(通过所在高校的“编委会”成员推荐)，经“编委会”认真评审，最后由清华大学出版

社审定出版。

目前，针对计算机类和电子信息类相关专业成立了两个“编委会”，即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。推出的特色精品教材包括：

（1）21世纪高等学校规划教材·计算机应用——高等学校各类专业，特别是非计算机专业的计算机应用类教材。

（2）21世纪高等学校规划教材·计算机科学与技术——高等学校计算机相关专业的教材。

（3）21世纪高等学校规划教材·电子信息——高等学校电子信息相关专业的教材。

（4）21世纪高等学校规划教材·软件工程——高等学校软件工程相关专业的教材。

（5）21世纪高等学校规划教材·信息管理与信息系统。

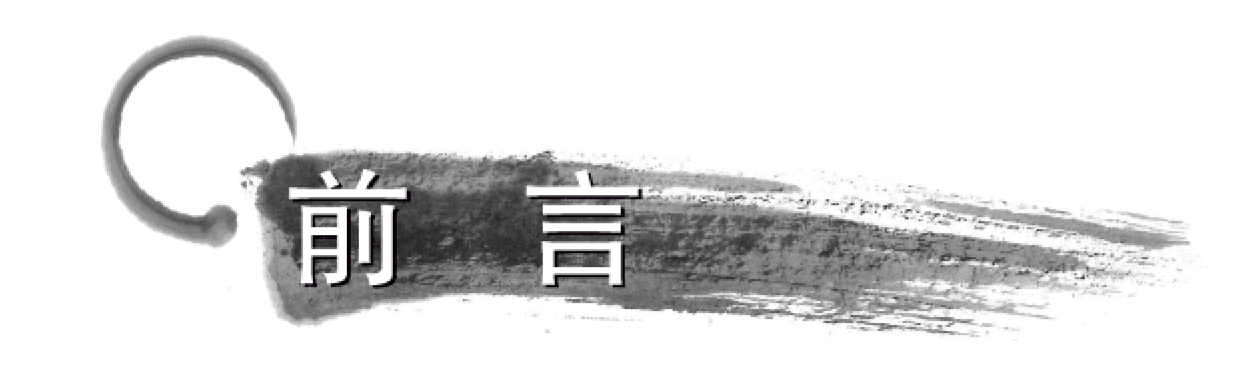
（6）21世纪高等学校规划教材·财经管理与应用。

（7）21世纪高等学校规划教材·电子商务。

（8）21世纪高等学校规划教材·物联网。

清华大学出版社经过三十多年的努力，在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌，为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格，这种风格将延续并反映在特色精品教材的建设中。

清华大学出版社教材编审委员会
联系人：魏江江
E-mail：weijj@tup.tsinghua.edu.cn



移动互联网作为目前备受瞩目的领域，已经成为各大企业争相发展的对象，可谓是企业必争之地。所谓移动互联网，实际上是移动网络和互联网融合的产物，它继承了移动网络随时随地随身和互联网分享、开放、互动的优势，是整合二者优势的"升级版本"，即运营商提供无线接入服务，互联网企业提供各种成熟的应用服务，可以这样说，移动互联网就是下一代互联网——Web 3.0。

互联网的发展经历了几个阶段：萌芽期、Web 1.0时代、Web 2.0时代。在互联网出现前，PC面临的最大问题是"孤芳自赏"，无法实现信息的共享，因此互联网应运而生。但是在萌芽期，人们慢慢发现，互联网只是局限在少数人，而它的潜力却不仅仅在此，因此，互联网开始出现在普通人的视线中，互联网进入Web 1.0时代。

在Web 1.0时代，最突出的问题是内容稀缺，因此，门户网站成为互联网的主流，如新浪、搜狐等网站开始崭露头角。接下来是被称为"信息爆炸"的阶段，互联网企业几乎把所有的报纸、杂志和一切可以搬上互联网的信息都搬上了互联网，这一时期互联网的最大问题不是信息过少，而是过多，造成信息的堆砌——用户需要的信息被大量杂乱的"垃圾"信息所淹没，在这样的形势下，信息的筛选和搜索成为核心要务。在这一时期，Google、百度等搜索引擎公司大行其道，成为互联网企业中的新宠。

随着互联网的进一步发展，人们渐渐发现，与以往任何媒体不同的是，互联网是一个极大的舞台，是一个人人都可以参与的舞台，而在这当中，"群众的力量"还远远未被挖掘出来。至此，Web 2.0悄然而至，社区、博客、C2C电子商务大行其道，它们共同的特点是：搭建一个平台，方便用户的参与——用户参与创建内容、提供信息、进行交易、进行传播。

从互联网的发展历程来看，互联网就是这样不断地"进化"的，而在进化的过程中，是围绕一个"中心"一个"特征"展开的——一个"中心"就是"以用户的需求"为中心，一个"特征"就是互联网特征：开放、平等、分享、互动、创新。互联网从萌芽期到Web 2.0时代，越来越开放，越来越平等，越来越强化用户的互动、分享和创新——说得大一点，互联网的发展过程就是人类解放的过程，人们通过互联网不断提高沟通效率，不断释放生产力。

我们有理由相信，互联网已开始和正在变革的阶段"移动互联网"——互联网与移动终端完美的融合，将互联网延伸至随时随地(Anytime，Anywhere)。互联网将不再局限于办公室或者家里的PC，而将延伸至PC和任何可移动终端，手机、PDA、MP3、手持游戏终端等——真正实现人类沟通和数字化生产的大解放。在这样的大背景下，多种移动计算平台蓬勃发展，经过最近四五年的竞争，Android平台逐渐成为该领域的佼佼者。处于这样一个阶段，我们更有理由去选择学习像Android这样优秀的移动计算平台，抓住移动互联网发展的机遇。

本书的目标是成为Android开发人员的"工具箱"，方便初步入门的读者进一步深化学习。全书对Android的一些基础知识仅进行粗略的介绍，而从一个较高的架构层次来介绍

Android 应用开发，并介绍了在开发过程中可能会常用到的一些较为高级功能。全书本着易学易用、“行重于知”原则进行编写，为此，书中使用了大量的精心编写的实例代码，这些代码注释详细，语句易懂，通过正文描述一步一步地引导读者掌握 Android 应用程序开发的方法和技巧。在使用本书的过程中，建议通过边学边实践的方式，一定要动手操作。书中所用的所有示例都是通过测试可以运行的，读者可从清华大学出版社网站下载。

全书共分为 12 章，各章内容要点如下：

第 1 章为 Android 简介，主要是介绍 Android 相关的一些背景常识，Android 的发展历史以及与其他移动操作系统的比较等，详细地列出了 Android 各个版本之间的异同，描绘出一幅 Android 发展的路线图。

第 2 章为 Android 系统结构，该部分讲解 Android 应用开发的体系结构，带领读者从根本上认识 Android 系统的本质，进一步介绍了 Android 核心 Linux 内核相关内容。

第 3 章为 Android 应用编程基础，考虑到面向的读者群应该已经具备了一些基础，该部分不包含相关开发环境的配置过程，而是进一步详细地讲解进行 Android 应用开发所需要掌握的基础知识，同时对相关所需的技能进行了说明。

第 4 章为用户界面开发，该部分内容对 Android 界面设计和实现进行了概括性的介绍。

第 5 章为数据与存储共享，该部分主要介绍如何在 Android 应用中保存和操作数据，重点在于如何操作 SQLite 关系数据库，难点是 ContentProvider 的理解和使用。

第 6 章为多进程与多线程，多线程多进程是一个相对较难的部分，重点介绍了 Android 的消息机制和进程间通信。

第 7 章为多媒体编程，内容包括音视频的播放与录制，动画效果的实现，双缓冲技术以及 2D 图形的绘制方法。

第 8 章为网络开发，该部分内容相对比较重要，主要包括了 Http 通信、Socket 通信、Web 服务的使用、WebView、Wi-Fi、蓝牙和 NFC 的内容，其中需要重点掌握的是 Http 通信、Socket 通信和 Web 服务。

第 9 章为 Android WebKit，该部分主要目的是让读者了解 WebKit 相关的内容。该章节的难点在于 WebKit 的结构比较复杂，要充分理解并不容易，因此请感兴趣的读者在学习时多花些时间。

第 10 章为 NDK 入门，NDK 在 Android 应用开发中属于比较少使用到的技能，但是在某些时候又十分必要，该部分主要内容是介绍如何正确地搭建好 NDK 的开发环境，之后再通过示例来验证开发环境。

第 11 章为游戏开发入门，实现了一个简单的俄罗斯方块游戏。

第 12 章为 Chrome 扩展，考虑到 Chrome 和 Android 之间的联系，该部分讲解了 Chrome 浏览器相关的知识，主要以理论讲解为主，并伴有相关的演示。

本书的完成体现了多人多年工作的积累。余堃对全书内容进行了统稿、修改、整理和定稿。其中第 1 章～第 8 章、第 10 章、第 11 章由段弘编写；第 9 章由史仁仁编写；第 12 章由柏露、张晏编写。唐雪飞负责全书的文字校对、源代码审查与整理工作。

本书在编写的过程中参考了相关文献，在此向这些文献的作者深表感谢。由于编者水平有限，书中难免有不妥之处，敬请专家和广大读者批评指正。在成书的过程中，感谢清华大学出版社在全书的技术准确性、编辑组织、文字润色等方面给予的帮助。

Android 应用开发是一门实践性很强的课程，相关的技能需要在 Android 应用开发的实践中去逐步掌握。由于 Android 应用程序开发所涉及的内容十分丰富，笔者很难也不可能在本书中穷尽所有的细节。不过笔者相信，当读者研读完本书之后，结合各自的实践经验，一定也会有很多的想法和感受，欢迎提出宝贵意见。

编　者

2012 年 10 月

目录

第1章 Android简介

1.1 Android简介

1.1.1 什么是Android

在日常的生活中，手机已经成为了人们必不可少的一件设备，基本上成为了人们随身携带的物品。而手机也从最开始只拥有相对单一功能(如语音通话、短信)的移动通信设备演变为拥有多种多样丰富功能的移动智能终端。在这样的发展情况下，智能手机操作系统相继诞生并蓬勃发展。而Android就是这些智能手机操作系统中的一员，它正凭借着优秀的性能和优良的特性吸引着越来越多的目光。

Android(英文音标：['ændrɔid])，中国大陆地区较多人使用“安卓”或“安致”，这个词汇最早出现于法国作家利尔亚当在1886年发表的科幻小说《未来夏娃》中，他将外表像人的机器起名为Android。现在提到的Android则是一种以Linux为基础的开放源码操作系统，主要用于便携式设备如手机、平板电脑。Android的创始人是Andy Rubin，最初属于Android公司的项目，只运行在手机平台上。Android公司于2005年由Google收购注资，并联合多家制造商组成了开放手机联盟(Open Handset Aillance)，从而借助这个开放联盟的资源进行开发和改良，使其逐渐扩展到平板电脑及其他领域。Android为了保障设备制造厂商的利益，提出了Android HAL架构图，HAL即Hardware Abstraction Layer——硬件抽象层，是介于硬件和执行于其上的软件之间的一种特殊软件。它的作用一方面是隐藏硬件方面的差异，并且将这些差异与操作系统核心相抽离，从而增进软件的可移植性；另一方面，由于HAL架构的存在，使得厂商可以不公布自身硬件的驱动程序源代码，从而保障了厂商的利益。而正由于Android的这种做法，被Linux社区认为违背了其开源的观念，2010年2月3日，Linux内核开发者Greg Kroah-Hartman将Android的驱动程序从Linux内核状态树(staging tree)上除去，从此，Android与Linux核心开发分道扬镳。

Android是Google于2007年11月5日发布的基于Linux开放性内核的开源手机操作系统，通常狭义地将Android理解为手机操作系统，而实际上Android由操作系统、中间件和关键的应用软件3部分组成，也就是一种称为软件堆叠(Software Stack)的架构，主要由3部分组成：底层以Linux内核工作为基础，由C语言开发，只提供基本功能；中间层包括函数库Library和虚拟机Virtual Machine(Dalvik)，由C++开发；最上层是各种应用软件，包括通话程序、短信程序等，应用软件可由各公司自行开发，主要以Java作为编程语言。自

2011年2月起，Android针对于平板计算机市场发布了3.x版本，该版本的推出使其不仅仅在手机领域表现优秀，同时也逐渐在平板计算机等移动设备领域中崭露头角。到2011年12月，Android又发布了4.0版本，该版本在覆盖范围上更进一步，目标是实现手机与平板计算机所使用操作系统的统一，4.0版本拥有了更加精致、简洁而美观的界面，同时也变得更加智能化(Android称其为beyond smart)。在随后将详细地介绍Android各版本的发展历史。

1.1.2 其他常见的移动操作系统

1. Symbian

Symbian OS(中文译“塞班操作系统”)是由诺基亚、索尼爱立信、摩托罗拉、西门子等几家大型移动通信设备商共同出资组建的一个合资公司，专门研发手机操作系统。Symbian操作系统的前身是EPOC，而EPOC是Electronic Piece of Cheese的首字母缩略词，其原意为“使用电子产品可以像吃乳酪一样简单”，这就是它在设计时所坚持的理念。它是一个实时性、多任务的纯32位操作系统，具有功耗低、内存占用少等特点，非常适合手机等移动设备使用，经过不断完善，可以支持GPRS、蓝牙、SyncML以及3G技术。更重要的是它是一个标准化的开放式平台，任何人都可以为支持Symbian的设备开发软件。与微软产品不同的是，Symbian将移动设备的通用技术，也就是操作系统的内核，与图形用户界面技术分开，能很好地适应不同方式输入的平台，也可以使厂商为自己的产品制作更加友好的操作界面，符合个性化的潮流，这也是用户能见到不同样子的Symbian系统的主要原因。

2. iOS

iOS是由苹果公司为其旗下的移动设备开发的操作系统。它主要是给iPhone、iPod touch、iPad等设备使用。与其基于的Mac OS X操作系统一样，它也是以Darwin为基础的。iOS的系统架构分为4个层次：核心操作系统层(the Core OS layer)、核心服务层(the Core Services layer)、媒体层(the Media layer)和Cocoa轻触层(the Cocoa Touch layer)。iPhone、iPod Touch和iPad使用基于ARM架构的中央处理器，而不是苹果的Mac计算机使用的x86处理器(就像以前的PowerPC或MC680x0)，它使用由PowerVR视屏卡渲染的OpenGL ES 1.1.。因此，Mac OS X上的应用程序不能直接复制到iOS上运行。它们需要针对iOS的ARM系统重新编写。但是借助于Safari浏览器能够支持“Web应用程序”的这种方式，能够实现广义上的应用程序“跨操作系统”。从iOS 2.0开始，通过审核的第三方应用程序已经能够通过苹果的App Store进行发布和下载了。遗憾的是，苹果至今仍没有宣布任何在iOS上运行Java的计划。

3. Palm OS

Palm OS是Palm公司的是一种32位的嵌入式操作系统，Palm OS是早期由U.S. Robotics(其后被3Com收购，再独立改名为Palm公司)研制的专门用于其掌上电脑产品Palm的操作系统。由于此操作系统完全为Palm产品设计和研发，而其产品由推出时就超过了苹果公司的Newton而获得了极大的成功，所以Palm OS也因此声名大噪。其后曾被

IBM、Sony、Handspring 等厂商取得授权，在旗下产品中使用。Palm OS 操作系统以简单易用为大前提，运作需求的内存与处理器资源较小，速度也很快；但不支持多线程，长远发展受到限制。它的操作界面采用触控式，差不多所有的控制选项都排列在屏幕上，使用触控笔便可进行所有操作。作为一套极具开放性的系统，开发商向用户免费提供 Palm 操作系统的开发工具，允许用户利用该工具在 Palm 操作系统的基础上编写、修改相关软件，使支持 Palm 的应用程序丰富多彩、应有尽有。Palm 操作系统最明显的优势还在于其本身是一套专门为掌上电脑编写的操作系统，在编写时充分考虑到了掌上电脑内存相对较小的情况，所以 Palm 操作系统本身所占的内存极小，基于 Palm 操作系统编写的应用程序所占的空间也很小，通常只有几十 KB，所以基于 Palm 操作系统的掌上电脑虽然只有几兆字节内存却可以运行众多的应用程序。Palm 在其他方面还存在一些不足，比如 Palm 操作系统本身不具有录音、MP3 播放功能等，如果需要使用这些功能，就需要另外加入第三方软件或硬件设备才可实现。

4. WebOS

WebOS 也是一个移动操作系统，它以 Linux 内核为主体，并加上部分 Palm 公司开发的专有软件。它主要是为 Palm 智能手机而开发。该平台于 2009 年 1 月 8 日的拉斯维加斯国际消费电子展首次向公众宣布，并于 2009 年 6 月 6 日发布。该平台事实上是 PalmOS 继任者，WebOS 将在线社交网络和 Web 2.0 一体化作为重点。WebOS 的图形用户界面是设计给带有触摸屏的手持设备使用的。它包括一系列应用程序，例如个人信息管理，主要使用 HTML5、JavaScript 及 CSS 进行开发。Palm 声称，设计围绕现有的技术以免开发者需学习一种新的编程语言。第一款搭载 WebOS 系统的智能手机是 Palm Pre，于 2009 年 6 月 6 日发售。由于 Palm 被 HP 收购，WebOS 被收归 HP 旗下。

2011 年 8 月 19 日凌晨，在惠普第三季度财报会议上，惠普宣布正式放弃围绕 TouchPad 平板电脑和 WebOS 手机的所有运营；2011 年 12 月 9 日，惠普新任总裁宣布 WebOS 开源，WebOS 的前景也成为了一个谜团。

5. Blackberry

Blackberry 是加拿大 RIM 公司推出的一种移动电子邮件系统终端，其中文名称为"黑莓"，它的特色是支持推动式电子邮件、手提电话、文字短信、互联网传真、网页浏览及其他无线资讯服务。从技术上来说，BlackBerry 是一种采用双向寻呼模式的移动邮件系统，兼容现有的无线数据链路。它出现于 1998 年，RIM 的品牌战略顾问认为，无线电子邮件接收器挤在一起的小小的标准英文黑色键盘，看起来像是草莓表面的一粒粒种子，于是起了这么一个有趣的名字。应该说，Blackberry 与桌面 PC 同步堪称完美，它可以自动把 Outlook 邮件转寄到 Blackberry 中，不过在用 Blackberry 发邮件时，它会自动在邮件结尾加上"此邮件由 Blackberry 发出"字样。其中 BlackBerry. enterpriseSolution 是一种领先的无线解决方案，可供移动专业人员用来实现与客户、同事和业务运作所需的信息连接。这是一种经证明有效的优秀平台，它为世界各地的移动用户提供了与大量业务信息和通信的安全的无线连接。

6. Linux

Linux 凭借其自由、免费、开放源代码的优势，经过来自互联网、遍布全球的程序员的努力，再加上 IBM、Sun 等计算机巨头的支持，在手机操作系统市场中异军突起，尤其是在众多知名厂商宣布支持 Linux 手机操作系统之后，Linux 的发展将不容忽视。由于 Linux 具有源代码开放、软件授权费用低、应用开发人才资源丰富等优点，便于开发个人和行业应用。这一点非常重要，因为丰富的应用是智能手机的优越性体现和关键卖点所在。从应用开发的角度看，由于 Linux 的源代码是开放的，有利于独立软件开发商(ISV)开发出硬件利用效率高、功能更强大的应用软件，也方便行业用户开发自己的安全、可控认证系统。

7. Windows Phone

Windows Phone 是由微软开发的移动操作系统，是之前 Windows Mobile 平台的继任者，但两者并不兼容。微软最早于 2010 年 2 月 15 日官方宣布推出了名为 Windows Phone 7 Series 的操作系统，之后缩减了名称正式命名为 Windows Phone 7。它的第一个完整 SDK 于 2010 年 9 月 16 日发布，随后微软正式发布了 10 款搭载 Windows Phone 7 系统的手机。2011 年 2 月 11 日，微软和诺基亚在伦敦召开新闻发布会，微软 CEO Steve Ballmer 和诺基亚 CEO Stephen Elop 共同宣布达成合作关系，Windows Phone 7 将成为诺基亚主要使用的智能手机操作系统，两家企业的目标是在建立起一个新的“全球手机生态系统”，意在与 Android、iOS 一起竞争智能手机操作系统市场。随后，微软和诺基亚对各自旗下所提供的一些服务进行了整合，例如在诺基亚的设备上将优先使用微软提供的 Bing 搜索引擎，诺基亚的地图服务和 Bing 地图服务进行了整合，诺基亚的 Ovi 商店也被整合到 Windows Phone Store 中。最早的一批诺基亚 Windows 手机在 2011 年 10 月召开的诺基亚世界大会上发布，包括 Lumia 800 和 Lumia 710，随后在 2012 年举办的消费电子产品展(Consumer Electronics Show)上诺基亚又发布了 Lumia 900。

2012 年 6 月 20 日，微软公开宣布了 Windows Phone 8，作为微软下一代的智能手机操作系统并将于 2012 年年底发布。值得注意的是，Windows Phone 8 将使用基于 Windows NT 内核的架构来替换之前所使用的基于 Windows CE 的架构，由于 Windows NT 与 Windows 8 共享了很多组件，因此应用程序可以很简单地在两者之间进行移植，不过这同时也意味着 Windows Phone 8 将不兼容 Windows Phone 7，微软和诺基亚也正式宣布 Windows Phone 8 将不能运行在之前发布的 Lumia 系列等用于运行 Windows Phone 7 的手机之上。

在市场占有量方面，根据 IDC(International Data Corporation，美国国际数据集团)的统计数字，Windows Phone(包括 Windows Mobile)从 2011 年 2 季度的 2.2%上涨到了 2012 年 2 季度的 3.5%。虽然目前占据市场份额仍然较小，但是随着微软的 Windows 8 和 Windows Phone 8 的推出，凭借微软强劲的实力，应该能够在未来的移动操作系统市场中有所作为。

1.1.3 Android 系统的优势

Android 平台具有如下一些优势。

1. 开放性

Android平台优势首先体现在它的开放性上，开放的平台允许任何移动终端厂商加入到Android联盟中来。显著的开放性可以使其拥有更多的开发者，随着用户和应用的日益丰富，一个崭新的平台也将很快走向成熟。开放性对于Android的发展而言，它有利于人气的积累，这里的人气包括消费者和厂商，而对于消费者来讲，最大的受益之处就是由开放性带来的丰富的软件资源。另外，开放的平台也会带来更大竞争，如此一来，消费者将可以用更低的价位购得心仪的手机。

2. 不再受限于通信运营商

在过去很长的一段时间，特别是在欧美地区，手机应用往往受到运营商制约，使用什么功能接入什么网络，几乎都受到运营商的控制。随着EDGE、HSDPA这些2G至3G移动网络的逐步过渡和提升，手机随意接入网络已经成为必然的模式。互联网巨头Google推动的Android终端天生就有网络特色，将让用户离互联网更近。

3. 丰富的硬件选择

这一点也是来自于Android平台的开放性。由于Android的开放性，众多的厂商会推出多种多样，功能各具特色的产品。这些不同产品功能上的差异和特色，却只会极小地影响到数据同步和软件的兼容，方便了用户的使用。

4. 不受任何限制的开发商

Android平台提供给第三方开发商一个十分宽泛、自由的环境，不会受到各种条条框框的阻挠，可想而知，会有多少新颖别致的软件会诞生。但也有其两面性，血腥、暴力、情色方面的程序和游戏如何控制正是留给Android难题之一。

5. 无缝结合的Google应用

如今叱咤互联网的Google已经走过10年的历程，从搜索巨人到全面的互联网渗透，Google服务如地图、邮件、搜索等已经成为连接用户和互联网的重要纽带，而Android平台手机将无缝结合这些优秀的Google服务。

1.2 Android发展历程

1.2.1 Android发展简史

在收购了Android之后，为了对Android进行推广，Google公司联合几十个手机相关企业组建了一个“开放手机联盟”，包括手机制造商人、手机芯片商、移动运营商等。开放手机联盟组建以后，Android就加速了它的发展进程，其主要发展历程如下：

(1) 2007年11月，34个手机相关企业共同联盟宣布成立了开放手机联盟。

(2) 2007年11月，在开放手机联盟宣布成立后的几天，Android发布了第一版本

的 SDK。

(3) 2008 年 4 月，为了推动 Android 的快速发展，手机联盟组织举办了一次 Android 开发竞赛，参赛者在规定的时间内，共提交了 1788 件作品，为 Android 的发展贡献了不可磨灭的力量。

(4) 2008 年 8 月，Android Market 正式上线，这是一个专供 Android 系统爱好者开发并出售其作品的开放平台，这一应用的使用，为 Android 的迅速发展积累了大量的应用。

(5) 2008 年 9 月，美国运营商 T-Mobile USA 在纽约正式发布了第一款基于 Android 操作系统的手机——T-Mobile G1。该手机是世界上第一部使用 Android 操作系统的手机，并且支持 Wi-Fi，支持 WCDMA/HSPA 网络，由宏达国际电子股份有限公司(简称 HTC 或宏达电)制造。

(6) 2008 年 10 月，Android 实现代码开源。

Android 开放源代码后，SDK 版本就开始了约每半年一个版本的快速更新。接下来将依次对这些版本的一些特性进行介绍。

1.2.2 Android SDK 版本发展及各版本新特性

Android 各版本对应的一些名称和编号及发行的时间等可于 SDK 文档的 reference/android/os/Build. VERSION_CODES. html 页面进行查看。值得一提的是，在上述页面中列出的一部分版本并没有公布出来，是由于很快就推出了它的下一个版本或者本身版本不稳定，这里仅简要介绍公开发布的几个稳定版本。

(1) Android 1. 0，第一版、最原始的 Android 操作系统，于 2008 年 10 月发布，版本代号为 BASE。

(2) Android 1. 1，于 2009 年 2 月发布，代号 BASE_1_1，该版本相对于 BASE 版本更新了部分 API，新增加一些功能，并修正了一些错误，同时增加了 com. google. android. maps 包，该包用于支持开发结合 Google 地图的应用。

(3) Android 1. 5，于 2009 年 3 月发布，代号 CUPCAKE，主要加入的新特性有：

- 提供屏幕虚拟键盘。
- 采用 WebKit 技术的浏览器。
- 使用 widgets 实现桌面个性化。
- 在线文件夹(Live Folder)快速浏览在线数据。
- 视频录制和分享。
- 图片上传。
- GPS 性能提高。
- 更快的标准兼容浏览器。
- Voice search 语音搜索。
- 支持立体声蓝牙耳机和免提电话。
- 改善了一些系统自带应用。

(4) Android 1. 6，于 2009 年 9 月发布，代号 DONUT，主要的更新如下：

- 完全重新设计的 Android Market，可以显示更多的屏幕截图。
- 手势支持，可以让开发者生成针对某个应用程序的手势库。

- 支持 CDMA 网络。
- TXT-To-Speech(TTS)系统,支持了更多语言的发音,包括英语、法语、德语、意大利语等。
- 快速搜索框,可直接搜索联系人、音乐、浏览历史、书签等手机内容。
- 全新的拍照界面:新版相机程序启动速度提高了 39%,拍照间的延迟减少了 28%。
- 应用程序耗电查看。
- 新增面向视觉或听觉困难人群的易用性插件。
- linux 内核升级到 2.6.29。
- 支持更多的屏幕分辨率,如 WVGA、QVGA 等。

(5) Android 2.1,于 2010 年 1 月发布,代号 ECLAIR,主要更新有:

- 优化硬件速度。
- 支持更多的屏幕分辨率。
- 改良的用户界面。
- 新的浏览器的用户接口和支持 HTML5。
- 新的联系人名单。
- 更好的白色/黑色背景比率。
- 改进 Google Maps 3.1.2。
- 支持 Microsoft Exchange。
- 支持内置相机闪光灯。
- 支持数码变焦。
- 改进的虚拟键盘。
- 支持蓝牙 2.1。
- 支持动态桌面的设计。

(6) Android 2.2,于 2010 年 6 月发布,代号 FROYO,主要的更新如下:

- 全面支持 Flash 10.1。
- 应用程序自动升级,让升级更加人性化。
- 支持应用程序安装在外置内存上。
- Linux 内核将升级为最新的 2.6.32 版本,系统更加稳定。
- 对系统性能进一步优化,让手机有更多的运行内存。
- 增加了对 3D 性能的优化,3D 性能更加强大。
- FM 功能也将在新系统中得到全面支持。

(7) Android 2.3,于 2010 年 12 月发布,主要更新内容如下:

- 用户界面更加美观。
- 提升游戏体验。
- 提升多媒体能力。
- 增加官方进程管理。
- 改善电源管理。
- 支持 NFC 近场通信。
- 全局下载管理。

- 全新虚拟键盘。
- 原生支持前置摄像头。
- 支持 SIP 网络电话。

(8) Android 3.x(3.0/3.1/3.2),于 2011 年 2 月 2 日发布,代号 Honeycomb,宣传是 Android 完全为 Tablet 而打造的系统版本,3.x 系列的 Android 针对于平板计算机做出了改造和优化,使其可以在平板计算机上展现更好的用户体验。有消息称之后用于手机端的版本与用于平板计算机的版本将分割为两个独立的分支,但目前情况还尚不明显。Honeycomb 的新特性如下:

- 专为平板计算机设计的 Android 版本。
- Google eBooks 上提供数百万本书。
- 支持平板计算机大屏幕、高分辨率。
- 新版 Gmail。
- Google Talk 视频通信功能。
- 3D 加速处理。
- 网页版 Market(Web store)详细分类显示,依个人 Android 分别设定安装应用程序。
- 新的短消息通知功能。
- 专为平板计算机设计的用户界面(重新设计的通知列与系统列)。
- 加强多任务处理的接口。
- 重新设计适用大屏幕的键盘及复制粘贴功能。
- 多个标签的浏览器以及私密浏览模式。
- 快速切换各种功能的相机。
- 增强的图库与快速滚动的联络人接口。
- 更有效率的 E-mail 接口。
- 支持多核心处理器。
- 3.2 优化 7 英寸平板显示。

(9) Android 4.0.x 版本,2011 年 10 月,Android 发布了 4.0 版本,版本代号为 IceCreamSandwich,该版本可以同时良好地工作在手机和平板计算机之上,实现了两种版本的融合和统一,截至目前(2012 年 1 月)最新发布的版本是 4.0.3(第二修订版),IceCreamSandwich 版本的新特性如下:

- 更加精致的 UI,这个版本对 UI 的提升是革命性的,其中变化较大的是 Action Bar(顶部的动作栏)以及对菜单的安排。
- 加强了对多任务执行的处理,在新的"正在运行程序列表"中可以同时观察多个正在运行的程序。
- 增加了 Home Screen 文件夹以及一个放置最喜爱程序的托盘。
- Widget 的尺寸可以灵活地调整。
- 全新的屏幕锁外观和解锁方式。
- 改进了文本输入的体验和文本拼写检查功能。
- 强大的语音输入功能。
- 新增对网络流量的监测和控制。

- 全新的联系人名片的组织方式，让用户更加方便地访问这些数据。
- 更加丰富的相机拍摄模式，并且增添了实时处理照片的功能。
- 与云端高度的整合。
- 面部识别解锁功能。
- 基于 NFC 的文件实时交换。
- 设备之间的 Wi-Fi 直连。

(10) Android 4.1.x 版本，在 2012 年 6 月 27 日召开的 Google I/O 大会上，Google 发布了 Android 4.1 版本，版本代号为 Jelly Bean，该版本基于 Linux kernel 3.0.31，其主要目标是提升系统的功能性和用户界面的性能，其中用户界面性能的提升借助了新提出的名为 Project Butter 的技术，该项技术凭借触摸预期、三重缓冲、扩展的垂直同步、60 帧的固定帧速率等策略来实现流畅的、"像黄油一样光滑"的用户界面。Jelly Bean 的源代码已经在 2012 年 7 月 9 日发布，第一台搭载 Jelly Bean 的设备——Nexus 7 平板于 2012 年 7 月 9 日发布。Jelly Bean 版本的新特性如下：

- 更加流畅的用户界面(Android 平台框架实现了所有绘图和动画间的垂直同步)。
- 增强的易使用性。
- 支持双向文本(RTL——从右到左，LTR——从左到右)。
- 扩展的通知栏。
- 桌面上的快捷方式和小部件能够自动调整位置和大小以适应新项目的添加。
- 使用蓝牙传输数据的 Android Beam 功能。
- 可离线工作的语音书写。
- 改进的语音搜索。
- 改进的相机应用。
- Google Wallet(Nexus 7 适用)。
- Google Now。
- 多通道音频。
- USB 音频。
- 音频无缝播放。
- 内置的浏览器被替换为 Android 移动版本的 Google Chrome 浏览器。

(11) Android 4.2 版本，该版本代号仍为 Jelly Bean。Google 原本计划于 2012 年 10 月 29 日在纽约市举行大会宣布该版本，但是由于美国部分地区遭遇飓风"桑迪"的袭击而取消了大会，取而代之的是在一场新闻发布会上宣布，该版本的口号是"一种全新口味的果冻豆(Jelly Bean)"，最先搭载该版本 Android 系统的设备包括 LG 的 Nexus 4(移动电话)以及三星的 Nexus 10(平板)，于 2012 年 11 月 13 日发布。Jelly Bean 4.2 版本的新特性包括：

- 新相机提供球面全景图拍摄功能。
- 具有手势输入功能的键盘。
- 更加强大的通知栏控制操作。
- "Daydream"屏幕保护程序。
- 多用户登录设备(仅限于 Tablet)。
- 支持无线显示。

- 提高的易访问性：通过三击屏幕可以放大整个屏幕，并且可以缩放屏幕进行操作，同时为盲人用户提供语音输出功能，以及通过 Gesture Mode 来操作 UI。
- 全新的时钟应用，内建世界时间、秒表、计时器等功能。
- 所有类型尺寸的设备使用统一的界面布局。
- 为更多的应用增加了一些扩展通知以及可操作的通知，允许直接在通知栏中处理和操作一些通知消息而不需要打开相应的应用。
- 安全增强式 Linux(SELinux)。
- 永久保持在线的 VPN(重启之后仍然保持连接)。

1.2.3 Android 前景展望

Android 的前景无疑是十分开阔的，随着越来越多的厂商加入由谷歌公司主导成立的开放手机联盟(OHA)，Android 的影响力已经越来越大，越来越多的人已经加入了开发 Android 的行列中，各种实用的手机软件也如雨后春笋般涌现，这些都保证了 Android 系统的良性发展，相信在不久 Android 将得到更多更广的应用。

参考文献

1. 维基百科 Android 词条：http://zh.wikipedia.org/wiki/Android.
2. Android 官方文档：http://developer.android.com/guide/basics/what-is-android.html.
3. 维基百科 Symbian 词条：http://zh.wikipedia.org/wiki/Symbian.
4. 维基百科 IOS 词条：http://zh.wikipedia.org/wiki/IOS.
5. 互动百科 Palm OS 词条：http://www.hudong.com/wiki/Palm+OS.
6. 互动百科 WebOS 词条：http://www.hudong.com/wiki/WebOS.
7. 百度百科 blackberry 词条：http://baike.baidu.com/view/88648.htm.
8. Android 的优势与不足：http://www.iteye.com/topic/855254.

第2章 Android 系统结构

2.1 Android 体系结构

为了降低一个大型的软件架构中的各个模块之间的耦合和相互依赖性，即提高组成这个结构的模块之间的正交性，通常会采用分层结构来实现这样的大型系统。

在设计良好的系统中，数据库代码与用户界面是正交的：可以改动界面而不用影响到数据库；也可以更换数据库而不用改动界面。一个典型的层次图如图 2-1 所示。

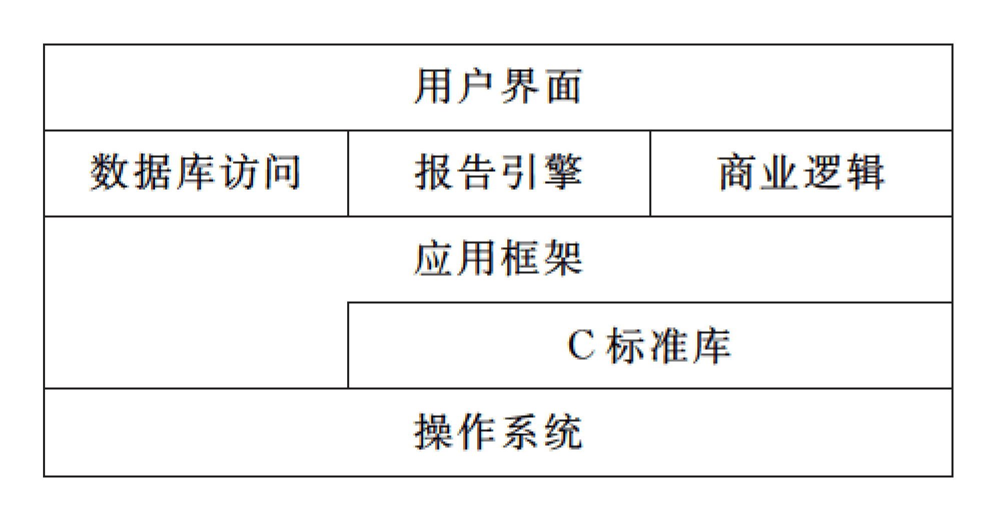


图 2-1 典型的层次图

类似地，Android 的系统结构也遵循了这个规则，采用了分层的结构，首先向读者展示一个框图，如图 2-2 所示，该图摘自 Android SDK 文档，它是用于说明 Android 系统体系结构最经典的一幅图，形象地描绘出了 Android 系统结构。

从图 2-2 中可以看出，Android 分为 4 层，从高到低分别是应用程序层、应用程序框架层、系统运行库层（包含函数库和 Android 运行时环境）和 Linux 内核层。

从广义上讲，Android 并不仅仅由一个操作系统组成，它还包括了一套中间件和应用程序的开发框架，因此我们通常也将 Android 称为一个“平台”。Android 从本质上来讲是一套软件的堆栈（Software Stack），主要分为 3 层，即操作系统、中间件和应用程序。其中，Android 的中间件可以再细分出两层：底层是函数库（Library）和虚拟机（Virtual Machine，VM），上层为应用程序框架（Application Framework）。图 2-2 的架构图中上方两部分（包括上两大层和 Android Runtime 中的 Core Libraries）使用 Java 语言开发，LIBRAIES 部分使用 C/C++ 开发，Linux Kernel 部分使用 C 开发，Android Runtime 下半部分为 Dalvik VM。Dalvik 虚拟机不同于我们熟知的 Hotspot 虚拟机（即通常意义上的 JVM），它是由 Android 另外提供的一个 Java 虚拟机，这也许是出于对 Java 授权方面的考虑，它支持已转换为.dex（即 Dalvik Executable）格式的 Java 应用程序的运行。.dex 格式是专为 Dalvik 设计的一种压缩格式，适合内存和处理器速度有限的系统。Dalvik 是由 Dan Bornstein 编写的，名字来源于他的祖先曾经居住过名叫 Dalvík 的小渔村，村子位于冰岛 Eyjafjörður。大多数虚拟机（包括 JVM）是一种堆栈机器，而 Dalvik 虚拟机则是基于寄存器的。两种架构各有优劣，一般而言，基于栈的机器需要更多指令，而基于寄存器的机器指令更大，详细的比较

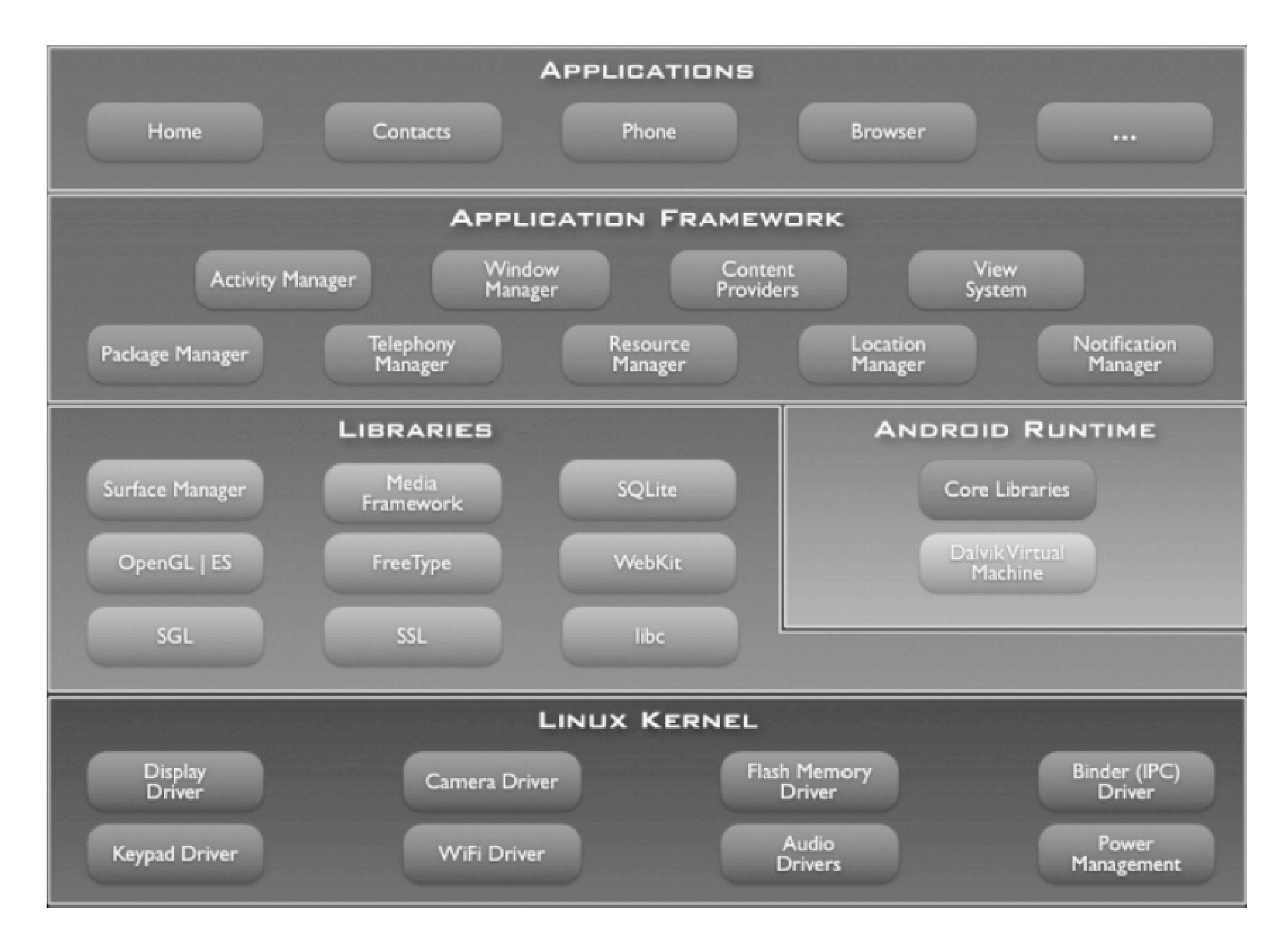


图 2-2 Android 系统结构图

将在后文中进行介绍。接下来从下到上逐层地对 Android 系统结构进行介绍。

2.1.1 内核层(Linux Kernel)

Android 平台的系统内核基于 Linux 2.6 内核,它是一个增强内核版本,其包含的主要功能有安全(Security)、内存管理(Memory Management)、进程管理(Process Managemen)、网络协议栈(Network Stack)、硬件驱动(Driver Model)等,Linux 内核同时也作为硬件和软件栈之间的抽象层,搭起了软件与硬件之间的桥梁,使得软件开发者不必关心内核的底层实现,而只需将精力全部投入到上层软件的开发中,而底层的工作都要由 Google 和手机开发商来完成,如驱动的更新、新硬件驱动的编写等。除了支持 Linux 内核本身所支持的一些设备驱动外,它还提供了用于支持 Android 平台的设备驱动,一些核心驱动主要包括:

- Android Binder,一个基于 OpenBinder 框架的驱动,用于对 Android 平台下的进程间通信提供支持。
- Android 电源管理(PM),一个基于标准 Linux 电源管理系统的轻量级的 Android 电源管理驱动,针对嵌入式设备做了大量的优化。
- 低内存管理器(Low Memory Killer),它相对于 Linux 标准 OOM(Out Of Memory)机制更加灵活,可以根据需要杀死进程从而为其他的进程释放其所需的内存。
- 匿名共享内存(Anonymous SHared MEMory,Ashmem),提供大块可共享的内存,这个功能使得进程间能够共享大块的内存,例如,借助这个功能,系统可以使用 Ashmem 保存一些图标,多个应用程序可以访问这个共享内存来获取图标。同时,它还为内核提供回收和管理这些共享内存的机制,即回收这些使用完的共享内存块的办法。如果一个进程试图访问这些已经被回收的内存块,将会得到一个错误的返

回值。这种机制使得它可以重新进行内存分配以及对数据进行初始化。

- Android PMEM(Physical),PMEM 用于向用户空间提供连续的物理内存区域,这是因为 DSP 和某些设备只能工作在连续的物理内存上。
- Android Logger,一个轻量级的日志设备,用于抓取系统产生的各种日志。
- Android Alarm,提供一个定时器,用于将设备从睡眠状态唤醒,同时提供了一个即使在设备睡眠时也会运行的时钟基准。
- USB Gadget 驱动,一个基于标准 Linux USB gadget 驱动框架的设备驱动。
- Android Ram Console and Log Device,这是 Android 为了调试的方便而添加的一个功能,它使得调试信息可以被写入到一个被称为 RAM Console 的设备中,它是一个基于 RAM 的 Buffer,此外,Android 还添加了一个独立的日志模块,使得用户空间的进程能够读写日志消息,以及调试打印信息等操作。
- Android timed device,提供了对设备进行定时控制的功能,例如对振动器或 LED 的控制。
- Android Debug Bridge,为了便于调试,Google 设计了这个调试工具,通常被称为"ADB",它使用 USB 作为连接方式,ADB 可以被看做是连接 Android 设备和 PC 的一套协议。

有关 Linux 内核的更多介绍,请读者阅读 2.2 节。

2.1.2　Android 运行时环境(Android Runtime)

Android 虽然使用 Java 程序语言来开发应用程序,但是却不是使用原有的 J2ME 版本来执行 Java 程序,而是采用 Android 自有的 Android Runtime 来执行。

Android Runtime 由下面两个核心部分组成:

(1) Core Libraries,即核心库,它实现了 Java 编程语言核心库的大多数功能。

(2) Dalvik Virtual Machine,相对于 Java 虚拟机(JVM),Android 实现了自己的虚拟机,即 Dalvik VM,不同于 JVM 所属于的堆栈结构机器(stack machine),Dalvik 属于寄存器机器(register machine),这两种类型的优劣在业界还是一个争执不下的论题,从技术层面来看,Register-based VM 的特性有个很大的好处,那就是对于目前主流的硬件架构,很容易与现有系统整合且达到最优化,而所需要的资源也相对较少。甚至在硬件实现上也比较容易。另外由于 Dalvik 并不是由 J2ME 实现,因此不存在 J2ME 授权相关的问题。对于每一个 Android 应用程序,它们都在自己的进程中运行,并拥有一个独立的 Dalvik 虚拟机实例。Dalvik 虚拟机使得一个设备可以同时高效地运行多个虚拟系统。Dalvik 虚拟机的许多地方参考了 Java 虚拟机的设计,Dalvik 虚拟机所执行的中间代码并非是 Java 虚拟机所执行的 Java Bytecode,同时也不直接执行 Java 类(Java Class File),而是依靠转换工具(dx.jar,3.1 节中将会提到)将 Java bytecode 转为 Dalvik VM 执行时特有的 dex(Dalvik Executable)格式。

通常来说,Java 的速度比较慢并不只是因为 Virtual Machine 的关系,Java 的程序编译成 Bytecode 也是关键因素之一,因为 Java VM 采用了 Stack-based 的方式来产生指令,所以所有的变量都需要 push 和 pop 操作,从而多出许多指令,而 Dalvik VM 所采用的 Register-based 方式,变量都存储在寄存器中,相比较而言,Dalvik VM 的指令就会少一点,速度也就

会较快一些。

Dalvik 虚拟机依赖于 Linux 内核的一些功能，如线程机制和底层内存管理机制。

2.1.3 函数库层(Libraries)

Android 包含了一些基础的 C/C++ 库，它们能被 Android 系统中不同的组件使用。它们通过 Android 应用程序框架为开发者提供服务。以下是一些核心库。

- Bionic System C library：一个从 BSD 继承来的标准 C 系统函数库(libc)，它是专门为基于嵌入式 Linux 的设备定制的。
- Media Libraries：基于 PacketVideo OpenCORE，该库支持多种常用的音频、视频格式回放和录制，同时支持静态图像文件。编码格式包括 MPEG4、H. 264、MP3、AAC、AMR、JPG、PNG 等。
- Surface Manager：提供对显示子系统的管理，并且为应用程序提供了 2D 图层和 3D 图层之间的无缝融合。
- LibWebCore：一个最新的 Web 浏览器引擎，支持 Android 浏览器及可嵌入应用程序的 Web 视图。
- SGL：底层的 2D 图形引擎。
- 3D Libraries：基于 OpenGL ES 1.0 APIs 实现，该库可以使用硬件 3D 加速(如果可用)或者使用高度优化的 3D 软加速。
- Free Type：位图(bitmap)和矢量(vector)字体显示。
- SQLite：SQLite 是一套开放源码的关系数据库，是一种对于所有应用程序可用并且功能强劲的轻型关系型数据库引擎。
- SSL：Secure Socket Layer，安全套接层，用于保护网页通信安全的协议。

另外，Android 还提供了一个硬件抽象层(Android Hardware Abstraction Layer，Android HAL)，它的主要目的是保护硬件提供商对其驱动程序的所有权。在前面已经提到，Android 并非把所有的设备驱动都放在 Linux 内核中，而是将其特有的设备驱动实现在用户空间层(userspace)，采用这种方式的主要原因是为了避开 Linux kernel 的 GPL license 中的相关条款约束：因为 Linux 是遵循 GPL 来发布的，也就意味着对 Linux 内核的任何修改都必须发布其源代码。采取 HAL 的方式就可以避开 GPL 条款，从而无须发布驱动的源代码，这样才能够维护硬件厂商的利益，毕竟对于硬件生产厂商来说，经济效益才是最重要的，如果他们的利益不能够得到保障，就很难说服他们加入到 Android 的阵营中来。Android 把控制硬件的操作都放到了用户空间层中，在 Linux 内核中的驱动只有最简单的读写寄存器的操作，去掉了各种功能性的操作(比如控制逻辑等)，这些能够体现硬件特性的操作都放到了 Android 的 HAL 层，而 Android 是基于 Apache 的许可协议，因此硬件厂商可以只提供二进制代码而不用发布源代码。

Android 的 HAL 的实现需要通过 JNI(Java Native Interface)，JNI 使得 Java 程序可以调用 C/C++写的动态链接库，采用这种技术，使得 HAL 可以用 C/C++语言来编写，从而实现较高的效率。Android 应用程序可以直接调用.so 库来使用硬件，也可以通过 app→app_manager→service(java)→service(jni)→HAL 的方式来使用。

实际上，可以将 HAL 作为单独的一层置于 Android 的系统结构中，这是由 Patrick

Brady 在 2008 年的 Google I/O 大会上提出的，这时候 Android 的系统结构图就变成了如图 2-3 所示的形式。

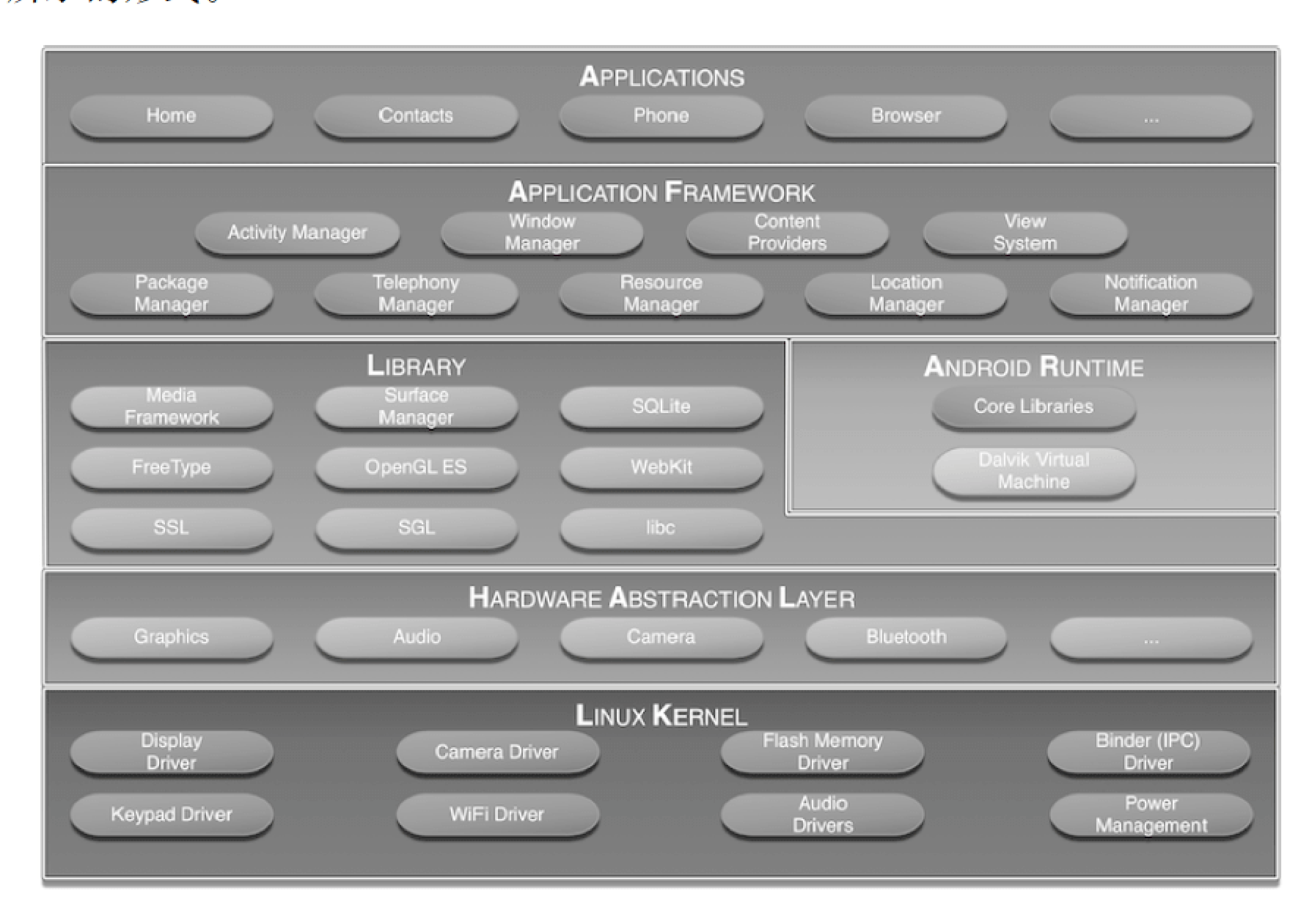


图 2-3　硬件抽象层(HAL)在 Android 系统结构所处的位置

2.1.4　应用程序框架层(Application Framework)

这一层是 Android 在编写随系统一起发布的核心应用时所使用的 API 框架，这个框架对于开发人员来说是完全可访问的，因此开发人员同样可以使用这些框架来开发自己的应用，这样就能够简化应用程序开发的架构设计，但是前提是必须遵守其框架的开发原则。这个应用程序的架构设计简化了组件的重用，它使得任何一个应用程序都可以发布它的功能模块，并且在 Android 的安全机制下，任何其他的应用程序都可以使用其所发布的功能模块。同样，该应用程序重用机制也使得用户可以方便地替换程序组件。

借助于 Android 提供的这个开放的开发平台，开发者能够开发出功能丰富并且富有创新性的应用，因为这个框架使得开发者能够自由而方便地使用设备所拥有的硬件、获取设备的地理位置信息、运行后台服务、设置定时器、在系统状态栏中添加消息提醒等，还有很多 Android 所提供的丰富特性。支撑应用程序正常运行的是一系列的服务和系统，具体包含如下几种。

- Views System(视图系统)：提供了丰富并且可扩展的视图组件，它们可以用于构建应用程序的视图，包括列表(lists)、网格(grids)、文本框(text boxes)、按钮(buttons)，甚至是嵌入式的 Web 浏览器(WebView)。
- Content Providers(内容提供器)：这个服务使得应用程序可以访问由另一个应用程序所维护的数据(例如访问由联系人应用所维护的联系人数据库)，或者向其他应用程序共享它们自己所维护的数据。
- Resource Manager(资源管理器)：提供应用程序对非代码资源的访问的方法，如本

地字符串、图形和布局文件(layout files)。

- Notification Manager(通知管理器):使应用程序可以在系统状态栏中显示提示信息。通知区域设定在手机的顶部,例如未读短信邮件、未接电话等通知消息都会在显示在顶部提示区域。
- Activity Manager(Activity 管理器):用于管理应用程序各 Activity 的生命周期并提供常用的导航回退功能。

2.1.5 应用程序层(Applications)

这一层通常指的是随 Android 平台一同发布的一系列核心应用程序,包括例如拨号程序、E-mail 客户端、SMS 短消息程序、日历、地图、浏览器、联系人管理程序等在内的智能手机必备的一些应用程序。所有的这些应用程序都是使用 Java 语言编写的。

2.2 Linux 内核简介

前面已经提到,Android 选择了 Linux 作为其内核基础,在其上做了少量的修改形成自己的内核。事实上,成熟的操作系统有很多,但是 Android 为什么会选择 Linux 内核呢?这与 Linux 的一些特性有关,比如 Linux 强大的内存管理和进程管理方案、基于权限的安全模式、支持共享库、经过认证的驱动模型以及 Linux 本身就是开源项目等。Android 选择了目前最稳定的 Linux 2.6 内核版本。

既然 Linux 内核对于 Android 如此重要,因此,为了让读者更加清楚地了解 Android,本节将对 Linux 内核做一些必要的介绍。

2.2.1 Linux 内核简介

Linux 是最受欢迎的自由计算机操作系统内核。它是一个用 C 语言写成,符合 POSIX 标准的类 UNIX 操作系统。Linux 最早是由芬兰黑客 Linus Torvalds 为尝试在 Intel x86 架构上提供自由免费的类 UNIX 操作系统而开发的。该计划开始于 1991 年,Linus Torvalds 当时在 Usenet 新闻组 comp. os. minix 所发表的贴子标志着 Linux 计划的正式开始。

在计划的早期有一些 Minix 黑客提供了协助,而今天全球无数程序员正在为该计划无偿提供帮助。从技术上说 Linux 是一个内核。"内核"指的是一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。一个内核不是一套完整的操作系统。一套基于 Linux 内核的完整操作系统叫做 Linux 操作系统或是 GNU/Linux。

Linux 是一个宏内核(monolithic kernel)系统。设备驱动程序可以完全访问硬件。Linux 内的设备驱动程序可以方便地以模块化(modularize)的形式设置,并在系统运行期间可直接装载或卸载。Linux 主要实现了进程管理(process management)、定时器(timer)、中断管理(interrupt management)、内存管理(memory management)、模块管理(module management)、虚拟文件系统接口(VFS layer)、文件系统(file system)、设备驱动程序(device driver)、进程间通信(inter-process communication)、网络管理(network management)、

系统启动(system init)等操作系统功能。

前面已经提到,Android 选择的 Linux 内核版本是 2.6 系列的,这里简要介绍一下 Linux 内核版本号的制定原则。Linux 的版本号分为两部分,即内核版本与发行版本。内核版本号由 3 个数字组成:r.x.y。

- r:目前发布的内核主版本。
- x:偶数表示稳定版本;奇数表示开发中版本。
- y:错误修补的次数。

一般来说,x 位为偶数的版本是一个可以使用的稳定版本,如 2.6.35;x 位为奇数的版本一般加入了一些新的内容,不一定很稳定,是测试版本,如 2.1.111。

表 2-1 列出了 Linux 内核版本的发展历程。

表 2-1 Linux 内核版本发展过程

内核版本号	时 间	内核发展史
0.00	1991.2—1991.4	两个进程分别显示 AAA BBB
0.01	1991.9	第一个正式向外公布的 Linux 内核版本
0.02	1991.10.5	Linus Torvalds 将当时最初的 0.02 内核版本发布到了 Minix 新闻组,很快就得到了反应。Linus Torvalds 在这种简单的任务切换机制上进行扩展,并在很多热心支持者的帮助下开发和推出了 Linux 的第一个稳定的工作版本
0.03	1991.10.5	常规更新,Bug 修复
0.10	1991.10	Linux 0.10 版本内核发布,0.11 版本随后在 1991 年 12 月推出,当时它被发布在 Internet 上,供人们免费使用
0.11	1991.12.8	基本可以正常运行的内核版本
0.12	1992.1.15	主要加入对数字协处理器的软件模拟程序
0.95(0.13)	1992.3.8	开始加入虚拟文件系统思想的内核版本
0.96	1992.5.12	开始加入网络支持和虚拟文件系统
0.97	1992.8.1	常规更新,Bug 修复
0.98	1992.9.29	常规更新,Bug 修复
0.99	1992.12.13	常规更新,Bug 修复
1.0	1994.3.14	Linux 1.0 版本内核发布,使用它的用户越来越多,而且 Linux 系统的核心开发队伍也建起来了
1.2	1995.3.7	常规更新,Bug 修复
2.0	1996.2.9	常规更新,Bug 修复
2.2	1999.1.26	常规更新,Bug 修复
2.4	2001.1.4	Linux 2.4.0 版本内核发布
2.6	2003.12.17	Linux 2.6 版本内核发布,与 2.4 内核版本相比,它在很多方面进行了改进,如支持多处理器配置和 64 位计算,它还支持实现高效率线程处理的本机 POSIX 线程库(NPTL)。实际上,性能、安全性和驱动程序的改进是整个 2.6.x 内核的关键
2.6.15	2006	Linux 2.6.15 版本内核发布。它对 IPv6 的支持在这个内核中有了很大的改进。PowerPC 用户现在有了一个用于 64 位和 32 位 PowerPC 的泛型树,它使这两种架构上的内核编辑成为可能

续表

内核版本号	时　　间	内核发展史
2.6.30	2009.6	改善了文件系统、加入了完整性检验补丁、TOMOYO Linux 安全模块、可靠的数据报套接字(datagram socket)协议支持、对象存储设备支持、FS-Cache 文件系统缓存层、nilfs 文件系统、线程中断处理支持等
2.6.32	2009.12	增添了虚拟化内存 de-duplicacion、重写了 writeback 代码、改进了 Btrfs 文件系统、添加了 ATI R600/R700 3D 和 KMS 支持、CFQ 低传输延迟时间模式、perf timechart 工具、内存控制器支持 soft limits、支持 S+Core 架构、支持 Intel Moorestown 及其新的固件接口、支持运行时电源管理以及新的驱动
2.6.34	2010.5	添加了 Ceph 和 LogFS 两个新的文件系统,其中前者为分布式的文件系统,后者是适用于 Flash 设备的文件系统。Linux Kernel 2.6.34 的其他特性包括新的 Vhost net、改进了 Btrfs 文件系统、对 Kprobes jump 进行了优化、新的 perf 功能、RCU lockdep、Generalized TTL Security Mechanism (RFC 5082) 及 private VLAN proxy arp (RFC 3069) 支持、asynchronous 挂起恢复等
2.6.36	2010.10	Tilera 处理器架构支持、新的文件通知接口 fanotify、Intel 显卡上实现 KMS 和 KDB 的整合、并行管理工作队列、Intel i3/5 平台上内置显卡和 CPU 的智能电源管理、CIFS 文件系统本地缓存、改善虚拟内存的层级结构,提升桌面操作响应速度、改善虚拟内存溢出终结器的算法、整合了 AppArmor 安全模型(注:与 SELinux 基于文件的标注不同,AppArmor 是基于路径的)

2.2.2 Linux 进程管理

Linux 是一种动态系统,能够适应不断变化的计算需求。Linux 计算需求的表现是以进程的通用抽象为中心的。进程可以是短期的(例如从命令行执行的一个命令),也可以是长期的(例如一种网络服务)。因此,对进程及其调度进行一般管理就显得极为重要。

在用户空间,进程是由进程标识符(PID)表示的。从用户的角度来看,一个 PID 是一个数字值,可唯一标识一个进程。一个 PID 在进程的整个生命期间不会更改,但 PID 可以在进程销毁后被重新使用,所以对它们进行缓存并不见得总是理想的。

在用户空间,创建进程可以采用几种方式。可以执行一个程序(这会导致新进程的创建),也可以在程序内调用一个 fork 或 exec 系统调用。fork 调用会导致创建一个子进程,而 exec 调用则会用新程序代替当前进程上下文。下面将对这几种方法进行讨论,以便读者很好地理解它们的工作原理。

下面首先展示进程的内核表示以及它们是如何在内核内被管理的,然后来看看进程创建和调度的各种方式(在一个或多个处理器上),最后介绍进程的销毁。

1. 进程在内核中的表示形式

在 Linux 内核内,进程是由相当大的一个称为 task_struct 的结构表示的。此结构包含

所有表示此进程所必需的数据,此外,还包含了大量的其他数据用来统计(accounting)和维护与其他进程的关系(父和子)。对 task_struct 的完整介绍超出了本文的范围,下段代码给出了 task_struct 的一小部分。这些代码包含了本文所要探索的这些特定元素。task_struct 位于"./linux/include/linux/sched.h"。如下代码是 task_struct 的一小部分:

```
struct task_struct {
    volatile long state;
    void * stack;
    unsigned int flags;
    int prio, static_prio;
    struct list_head tasks;
    struct mm_struct * mm, * active_mm;
    pid_t pid;
    pid_t tgid;
    struct task_struct * real_parent;
    char comm[TASK_COMM_LEN];
    struct thread_struct thread;
    struct files_struct * files;
    ...
};
```

在上面列出的代码片段中,可以看到几个比较显而易见的变量,比如执行的状态、堆栈、一组标志、父进程、执行的线程(可以有很多)以及开放文件。稍后会对其进行详细说明,这里只简单加以介绍。state 变量是一些表明任务状态的比特位。最常见的状态有:TASK_RUNNING 表示进程正在运行,或是排在运行队列中正要运行;TASK_INTERRUPTIBLE 表示进程正在休眠、TASK_UNINTERRUPTIBLE 表示进程正在休眠但不能叫醒;TASK_STOPPED 表示进程停止等。这些标志的完整列表可以在./linux/include/linux/sched.h 内找到。

flags 定义了很多指示符,表明进程是否正在被创建(PF_STARTING)或退出(PF_EXITING),或是进程当前是否在分配内存(PF_MEMALLOC)。可执行程序的名称(不包含路径)占用 comm(命令)字段。

每个进程都会被赋予优先级(称为 static_prio),但进程的实际优先级是基于加载以及其他几个因素动态决定的。优先级值越低,实际的优先级越高。

tasks 字段提供了链接列表的能力。它包含一个 prev 指针(指向前一个任务)和一个 next 指针(指向下一个任务)。

进程的地址空间由 mm 和 active_mm 字段表示。mm 代表的是进程的内存描述符,而 active_mm 则是前一个进程的内存描述符(为改进上下文切换时间的一种优化)。

thread_struct 则用来标识进程的存储状态。此元素依赖于 Linux 在其上运行的特定架构,在 ./linux/include/asm-i386/processor.h 内有这样的一个例子。在此结构内,可以找到该进程自执行上下文切换后的存储(硬件注册表、程序计数器等)。

2. 进程创建

在很多情况下,进程都是动态创建并由一个动态分配的 task_struct 表示。一个例外是 init 进程本身,它总是存在并由一个静态分配的 task_struct 表示。在 ./linux/arch/i386/

kernel/init_task.c 内可以找到这样的一个例子。

Linux 内所有进程的分配有两种方式：第一种方式是通过一个哈希表，由 PID 值进行哈希计算得到；第二种方式是通过双链循环表。循环表非常适合于对任务列表进行迭代。由于列表是循环的，没有头或尾；但是由于 init_task 总是存在，所以可以将其用做继续向前迭代的一个锚点。

下面介绍如何从用户空间创建一个进程。用户空间任务和内核任务的底层机制是一致的，因为二者最终都会依赖于一个名为 do_fork 的函数来创建新进程。在创建内核线程时，内核会调用一个名为 kernel_thread 的函数（参见 ./linux/arch/i386/kernel/process.c），此函数执行某些初始化后会调用 do_fork。

创建用户空间进程的情况与此类似。在用户空间，一个程序会调用 fork，这会导致对名为 sys_fork 的内核函数的系统调用（参见 ./linux/arch/i386/kernel/process.c）。函数关系如图 2-4 所示。

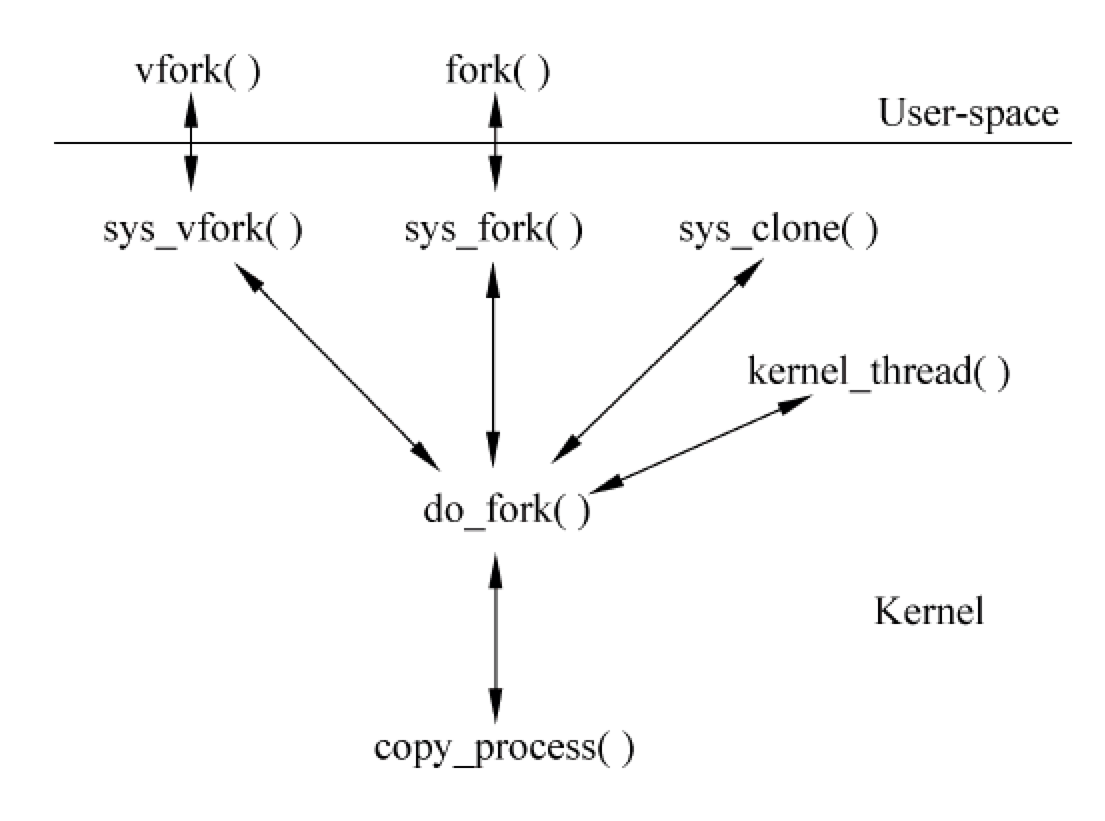


图 2-4 负责创建进程的函数的层次结构

从图 2-4 中可以看到，do_fork 是进程创建的基础。可以在 ./linux/kernel/fork.c 内找到 do_fork 函数（以及合作函数 copy_process）。

do_fork 函数首先调用 alloc_pidmap，该调用会分配一个新的 PID。接下来，do_fork 检查调试器是否在跟踪父进程。如果是，在 clone_flags 内设置 CLONE_PTRACE 标志以做好执行 fork 操作的准备。之后 do_fork 函数还会调用 copy_process，向其传递这些标志、堆栈、注册表、父进程以及最新分配的 PID。

新的进程在 copy_process 函数内是作为父进程的一个副本创建的。此函数能执行除启动进程之外的所有操作，启动进程在之后进行处理。copy_process 内的第一步是验证 CLONE 标志以确保这些标志是一致的。如果不一致，就会返回 EINVAL 错误。接下来，询问 Linux Security Module (LSM) 看当前任务是否可以创建一个新任务。要了解有关 LSM 在 Security-Enhanced Linux (SELinux) 上下文中的更多信息，请参见本章参考文献第 7 条。

接下来，调用 dup_task_struct 函数（在 ./linux/kernel/fork.c 内），这会分配一个新 task_struct 并将当前进程的描述符复制到其内。在新的线程堆栈设置好后，一些状态信息也会被初始化，并且会将控制返回给 copy_process。控制回到 copy_process 后，除了其他

几个限制和安全检查之外，还会执行一些常规管理，包括在新 task_struct 上的各种初始化。之后，会调用一系列复制函数来复制此进程的各个方面，比如复制开放文件描述符(copy_files)、复制符号信息(copy_sighand 和 copy_signal)、复制进程内存(copy_mm)以及最终复制线程(copy_thread)。

之后，这个新任务会被指定给一个处理程序，同时对允许执行进程的处理程序进行额外的检查(cpus_allowed)。新进程的优先级从父进程的优先级继承后，执行一小部分额外的常规管理操作，而且控制也会被返回给 do_fork。在此时，新进程存在但尚未运行。do_fork 函数通过调用 wake_up_new_task 来修复此问题。此函数(可在 ./linux/kernel/sched.c 内找到)初始化某些调度程序的常规管理信息，将新进程放置在运行队列之内，然后将其唤醒以便执行。最后，一旦返回至 do_fork，此 PID 值即被返回给调用程序，进程完成。

3. 进程调度

操作系统要实现多进程，进程调度是一个关键点。进程调度就是对当前处于“正在运行”状态的一些进程进行调度。存在于 Linux 的进程通过 Linux 调度程序被调度。Linux 调度程序维护了针对每个优先级别的一组列表，其中保存了 task_struct 引用。任务通过 schedule 函数(在./linux/kernel/sched.c 内)调用，它根据加载及进程执行历史决定最佳进程。

操作系统为了协调多个进程的“同时”运行，最基本的手段就是给进程定义优先级。定义了进程的优先级，如果有多个进程同时处于可执行状态，那么优先级高的进程就将被优先执行。那么，进程的优先级该如何确定呢？Linux 提供了两种方式：由用户程序指定、由内核的调度程序动态调整。

Linux 内核将进程分成两个级别：普通进程和实时进程。实时进程的优先级都高于普通进程，这两种进程的调度策略也有所不同，首先来介绍一下实时进程的调度策略。

1) 实时进程的调度策略

所谓“实时”，其基本含义是“给定的操作一定要在确定的时间内完成”。重点并不在于操作一定要处理得多快，而是时间要可控(在最坏情况下也能够在给定的时间内完成)。这样的“实时”被称为“硬实时”，多用于很精密的系统之中(比如火箭、导弹之类的)。一般来说，硬实时的系统是相对比较专用的。Linux 实际上无法满足这样的要求，中断处理、虚拟内存等机制的存在给处理时间带来了很大的不确定性。硬件的 cache、磁盘寻道、总线争用都会带来不确定性。事实上，像 Linux 这样号称实现了“实时”的通用操作系统，其实只是实现了“软实时”，即尽可能地满足进程的实时需求。软实时的含义是：如果一个进程有实时需求(它是一个实时进程)，则只要它是可执行状态的，内核就一直让它执行，以尽可能地满足它对 CPU 的需要，直到它完成所需要做的事情，然后睡眠或退出(变为非可执行状态)。而如果有多个实时进程都处于可执行状态，则内核会先满足优先级最高的实时进程对 CPU 的需要，直到它变为非可执行状态。于是，只要高优先级的实时进程一直处于可执行状态，低优先级的实时进程就一直不能得到 CPU；只要一直有实时进程处于可执行状态，普通进程就一直不能得到 CPU。对于实时进程，内核不会试图调整其优先级。因为进程是否实时或者其实时的程度有多大这些问题都与用户程序的应用场景相关，只有用户能够回答，内核不能臆断。综上所述，实时进程的调度是非常简单的，因为进程的优先级和调度策略都由用户定死了，内核只需要总是选择优先级最高的实时进程来调度执行即可。

2）普通进程的调度策略

实时进程调度的中心思想是，让处于可执行状态的最高优先级的实时进程尽可能地占有CPU，因为它有实时需求；而普通进程则被认为是没有实时需求的进程，于是调度程序力图让各个处于可执行状态的普通进程和平共处地分享CPU，从而让用户觉得这些进程是同时运行的。

与实时进程相比，普通进程的调度要复杂得多。内核需要考虑两个相对复杂的问题。

（1）动态调整进程的优先级。

按进程的行为特征，可以将进程分为“交互式进程”和“批处理进程”：

- 交互式进程（如桌面程序、服务器等）主要的任务是与外界交互。这样的进程应该具有较高的优先级，它们总是睡眠等待外界的输入。而在输入到来，内核将其唤醒时，它们又应该很快被调度执行，以做出响应。比如一个桌面程序，如果鼠标点击后半秒钟还没反应，用户就会感觉系统“卡住”了。
- 批处理进程（如编译程序）主要的任务是做持续的运算，因而它们会持续处于可执行状态。这样的进程一般不需要高优先级，比如编译程序多运行了几秒钟，用户多半不会太在意。

如果用户能够明确知道进程应该有怎样的优先级，可以通过nice、setpriority系统调用来对优先级进行设置（如果要提高进程的优先级，要求用户进程具有CAP_SYS_NICE能力）。

然而应用程序未必就像桌面程序、编译程序这样典型。程序的行为可能五花八门，可能一会儿像交互式进程，一会儿又像批处理进程，以至于用户难以给它设置一个合适的优先级。

再者，即使用户明确知道一个进程是交互式还是批处理，也多半碍于权限或因为偷懒而不去设置进程的优先级。

于是，最终区分交互式进程和批处理进程的重任就落到了内核的调度程序上。

调度程序关注进程近一段时间内的表现（主要是检查其睡眠时间和运行时间），根据一些经验性的公式，判断它现在是交互式的还是批处理的，程度如何等。最后决定给它的优先级做一定的调整。

进程的优先级被动态调整后，就出现了两个优先级：

① 用户程序设置的优先级（如果未设置，则使用默认值），称为静态优先级。这是进程优先级的基准，在进程执行的过程中往往是不改变的。

② 优先级动态调整后，实际生效的优先级。这个值是可能时时刻刻都在变化。

（2）调度的公平性。

在支持多进程的系统中，理想情况下，各个进程应该是根据其优先级公平地占有CPU。而不会出现某几个进程长时间占有CPU这样的不可控的情况。

Linux实现公平调度基本上是两种思路：

① 给处于可执行状态的进程分配时间片（按照优先级），用完时间片的进程被放到“过期队列”中。等可执行状态的进程都过期了，再重新分配时间片。

② 动态调整进程的优先级。随着进程在CPU上运行，其优先级被不断调低，以便其他优先级值较低的进程得到运行机会。

后一种方式有更小的调度粒度，并且将“公平性”与“动态调整优先级”两件事情合而为一，大大简化了内核调度程序的代码。因此，这种方式也成为内核调度程序的新宠。

对于进程调度策略来说，还需要考虑调度程序的效率、调度触发时机的选择、内核抢占、多处理器下的负载均衡、优先级继承等方面的因素，限于篇幅本书中就不对这些进行介绍，读者可以翻阅相关的 Linux 书籍获取更多的信息。

2.2.3　Linux 文件系统

文件系统是对一个存储设备上的数据和元数据进行组织的机制。这个定义非常宽泛，因为文件系统是一个很大的系统结构，实在不是能够用一两句话就能够简单地进行概括的。目前业界对文件系统存在着诸多定义，这些定义各有特点，但基本上本质上是一致的，作为示例，下面列出一些对文件系统的定义：

- 文件系统是包括在一个磁盘（包括光盘、软盘、闪盘及其他存储设备）或分区的目录结构；一个可应用的磁盘设备可以包含一个或多个文件系统；如果想进入一个文件系统，首先要做的是挂载（mount）文件系统；为了挂载（mount）文件系统，必须指定一个挂载点；一旦文件系统被挂载，就可通过这个挂载点对这个文件系统进行访问。
- 文件系统是在一个磁盘（包括光盘、软盘、闪盘及其他存储设备）或分区组织文件的方法，如 NTFS 或 FAT。
- 文件系统是基于操作系统的，建立在磁盘媒质上的可见体系结构。
- 文件系统是文件的数据结构或组织方法。在 UNIX 中，文件系统涉及两个非常独特的事情，目录树或在磁盘或分区上文件的排列。
- 文件系统是有组织存储文件或数据的方法，目的是易于查询和存取。文件系统是基于一个存储设备，比如硬盘或光盘，并且包含文件物理位置的维护；也可以说文件系统也是虚拟数据或网络数据存储的方法，比如 NFS。
- 文件系统是基于被划分的存储设备上的逻辑上单位上的一种定义文件的命名、存储、组织及取出的方法。

在 Linux 中常用的文件系统主要有 ext3、ext2 及 reiserfs；Windows 和 DOS 常用的文件系统是 FAT 系列（包括 FAT16 及 FAT32 等）和 NTFS 文件系统；光盘文件系统是 ISO-9660 文件系统；网络存储 NFS 服务器在客户端访问时，文件系统是 NFS。

2.2.4　Linux 线程管理

Linux 的线程是轻量级线程（Light Weight Process，LWP），在 Linux 中，它的行为以及管理调度方式非常类似于进程。在线程概念出现以前，为了减小进程切换的开销，操作系统设计者逐渐修正进程的概念，逐渐允许将进程所占有的资源从其主体剥离出来，允许某些进程共享一部分资源，例如文件、信号，数据内存，甚至代码，这就发展出轻量进程的概念。Linux 内核在 2.0.x 版本就已经实现了轻量进程，应用程序可以通过一个统一的 clone()系统调用接口，用不同的参数指定创建轻量进程还是普通进程。在内核中，clone()调用经过参数传递和解释后会调用 do_fork()，这个核内函数同时也是 fork()、vfork()系统调用的最

终实现。

由于 Linux 线程与进程十分相似，其他相关的信息请参考前面 2.2.2 节的相关内容。

2.2.5 Linux 内存管理

内存是 Linux 内核所管理的最重要的资源之一，内存管理系统是操作系统中最为重要的部分。Linux 内存管理建立在基本的分页机制基础上，在 Linux 内核中 RAM 的某些部分将会永久地分配给内核，并用来存放内核代码以及静态内核数据结构。RAM 的其余部分称为动态内存，这不仅是进程所需的宝贵资源，也是内核本身所需的宝贵资源。实际上，整个系统的性能取决于如何有效地管理动态内存。因此，现在所有多任务操作系统都在经历优化对动态内存的使用，也就是说，尽可能做到当要时分配，不需要时释放。

内存管理是操作系统中最复杂的管理机制之一。Linux 中采用了很多有效的管理方法，包括页表管理、高端内存(临时映射区、固定映射区、永久映射区、非连续内存区)管理、为减小外部碎片的伙伴系统、为减小内部碎片的 slab 机制以及紧急内存管理等。

每个程序员都希望拥有无穷大并且快速的内存，为了达到无穷大的目标，Linux 引入了虚拟存储系统，为了达到快速的目标，Linux 又引入了 cache、交换机制等技术，从而使得内存在容量上接近硬盘，在速度上接近 cache。Linux 的内存管理采取的是分页机制。它的设计目的是分时多任务。Linux 可同时处理 256 个任务，同时它采用了两级饱和机制来分别内核进程与用户进程。

Linux 虚拟内存的实现，需要通过如下几种机制来实现：

- 地址映射机制。
- 内存的分配与回收。
- 请页机制。
- 交换机制。
- 内存共享机制。

进程是操作系统分配内存的最小单位，所有进程(执行的程序)都必须占用一定数量的内存，它或是用来存放从磁盘载入的程序代码，或是存放取自用户输入的数据等。不过进程对这些内存的管理方式因内存用途不一而不尽相同，有些内存是事先静态分配和统一回收的，而有些却是按需要动态分配和回收的。

任何一个常规的进程都会涉及 5 种不同的数据段。下面简单归纳一下进程对应的内存空间中所包含的 5 种不同的数据区。

- 代码段：代码段是用来存放可执行文件的操作指令，也就是说，它是可执行程序在内存中的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，而不允许写入(修改)操作——它是不可写的。
- 数据段：数据段用来存放可执行文件中已初始化全局变量，换句话说就是存放程序静态分配的变量和全局变量。
- BSS 段：BSS 段包含了程序中未初始化的全局变量，在内存中，bss 段全部置零。
- 堆(heap)：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上(堆被扩张)；当利用 free 等函数释放内存时，被释放的内存从堆中被剔

除(堆被缩减)。

- 栈：栈是用户存放程序临时创建的局部变量,也就是说,函数括号"{}"中定义的变量(但不包括 static 声明的变量,static 意味着在数据段中存放变量)。除此以外,在函数被调用时,其参数也会被压入发起调用的进程栈中,并且待到调用结束后,函数的返回值也会被存放回栈中。由于栈的先进先出特点,所以栈特别方便用来保存/恢复调用现场。从这个意义上讲,可以把堆栈看成一个寄存、交换临时数据的内存区。

进程内存管理的对象是进程线性地址空间上的内存镜像,这些内存镜像其实就是进程使用的虚拟内存区域(memory region)。进程虚拟空间是个 32 位或 64 位的"平坦"(独立的连续区间)地址空间(空间的具体大小取决于体系结构)。要统一管理这么大的平坦空间可绝非易事,为了方便管理,虚拟空间被划分为许多大小可变的(但必须是 4096 的倍数)内存区域,这些区域在进程线性地址中像停车位一样有序排列。这些区域的划分原则是"将访问属性一致的地址空间存放在一起",所谓访问属性指的就是"可读、可写、可执行"。

如果要查看某个进程占用的内存区域,可以使用命令 cat /proc/maps。

参考文献

1. Android Hunt / David Thomas,程序员修炼之道,北京:电子工业出版社,2004.
2. Android developers Dev Guide - What is Android: http://developer.android.com/guide/basics/what-is-android.html.
3. Android 内核和驱动程序: http://hi.baidu.com/eastream/blog/item/d425bd8997e86286a5c2722f.html.
4. Patrick Brady dissects Android: http://www.zdnet.com/blog/burnette/patrick-brady-dissects-android/584.
5. 维基百科 Linux 内核词条: http://zh.wikipedia.org/zh/Linux%E5%86%85%E6%A0%B8.
6. Linux 进程管理剖析: http://www.ibm.com/developerworks/cn/linux/l-linux-process-management/.
7. 安全增强 Linux(SELinux)剖析: http://www.ibm.com/developerworks/cn/linux/l-selinux.

第3章 Android 应用编程基础

3.1 Android SDK

SDK——Software Development Kit，即软件开发工具包，通常包含了一系列的软件开发工具。这些开发工具通常由特定的厂商或者组织提供，用于开发特定的软件包、运行在特定的软件框架上的软件、运行在特定的硬件平台或者计算机系统上的软件等，又或者是用于开发运行在特定的游戏机上的游戏、运行在特定的操作系统或者其他平台上的软件。它或许只是简单地为某个程序设计语言提供应用程序接口的一些文件，但也可能包括能与某种嵌入式系统通信的复杂的硬件。一般的工具包还包括用于调试和其他用途的实用工具。SDK 还经常包括示例代码、支持性的技术注解或者其他的为基本参考资料澄清疑点的支持文档（请参见维基百科）。

同样，为了提供应用开发的支持，Android 也为开发者提供了开发工具包即 Android SDK，Android SDK 包含了一套较为全面的开发工具，并且随着 SDK 版本的更新，这个工具包还在进一步地丰富。它主要包括调试器、应用程序打包工具以及其他众多协助开发和设计的工具、Java 类库、基于 QEMU 的模拟器、API 文档、示例代码以及开发指南。读者可以在 SDK 安装完成后到相应的安装目录下查看，并且对该目录下的每一个文件进行归类，看看它们分别属于上面所说的哪一种类型。

Android SDK 需要工作在运行了特定操作系统的计算机平台之上，目前已经支持的操作系统有 Linux（包括绝大部分流行的 Linux 桌面发行版）、Mac OS X 10.4.9 及以上版本以及 Windows XP 及以上版本的操作系统。同时，Android SDK 的使用还需要借助于一个集成开发环境（Integrated Development Environment，IDE），现在官方提供支持的并且推荐使用的开发环境是 Eclipse，在 Eclipse 中还需要安装由 Android 提供的插件 ADT（Android Development Tools，Android 开发工具），ADT 除了具有将 Eclipse 与 Android SDK 建立起联系的功能之外，还提供了一些用于辅助开发和调试的小工具，例如 LogCat（用于显示日志信息）、File Explorer（文件浏览器，可浏览模拟器或者真机上的文件）、Devices（设备管理器）、Emulator Control（模拟器控制器）、Layout View（布局查看器）等非常实用的小工具，可以将 ADT 理解为 Eclipse 与 Android SDK 之间的桥梁，有了 ADT，就能够在 Eclipse 中方便地使用由 Android SDK 提供的一些功能了。当然，如果擅长于不借助于集成开发环境的开发方法，也可以使用任何自己喜欢的文本编辑器来编写代码和 xml 文件，然后使用命令行工具对代码进行编译和构建（借助于 JDK 和 Apache Ant 等工具），也可以在命令行环

境下对连接的 Android 设备(模拟器或真机)进行远程管理。

Android SDK 与 Android 平台版本的发布历史是同步的,因此,如果所开发的应用需要运行在安装了较早版本 Android 平台的设备上,就可以采用在较早版本的 SDK 下开发应用程序的方式来实现这个目的。同样地,可以同时使用多个版本的 platform 和 tools 来对应用程序进行测试,从而对应用的兼容性进行评估。

借助于 Android SDK、ADT 和 Eclipse,可以方便地将所开发的应用程序打包成.apk 文件发布,使得其他任何 Android 设备上都能够安装该应用程序。.apk 格式的安装文件在安装完成后将被以 package 的形式存放在/data/app 文件夹下,package 中包含了已经编译好的可执行的.dex 文件以及其他的资源文件等。

要安装 SDK 以及完整地搭建好 Android 应用开发环境,读者可以访问 Android 官方网站:http://developer.android.com,然后跟随上面的安装指南一步一步地进行安装,安装过程比较简单,此处不再赘述。

一般情况下,我们都使用 Android SDK 来开发应用程序,Android SDK 也能够满足绝大部分的开发需求,如果确实有一些功能无法通过 SDK 来实现,可以考虑使用 3.2 节介绍的另外一套开发工具包——NDK。不过这里提醒读者:NDK 并不是一个万能良药,除非必须使用它,否则还是首先考虑使用 SDK 进行开发。

3.2　Android NDK

NDK——Native Development Kit,即本地开发工具包,它使得应用程序可以部分使用本地代码编写(通常使用 C 代码),NDK 能够将这部分代码编译为可供 JNI 调用的库,从而使得上层 Java 代码能够调用这部分由本地代码编写的方法,利用 JNI 甚至还可以在 Java 与本地代码之间传递对象,由于 NDK 开发不同于 SDK,因此本书的第 11 章专门介绍更多详细的 NDK 入门知识。

3.3　Android 应用执行环境的特点

Android 作为一种运行在移动设备和嵌入式设备中的操作系统。受到这种特定环境的限制,所要考虑的因素也就比通常在 PC 上开发应用程序时有所不同,本节就将具体地对 Android 应用执行环境的特点进行介绍,读者在开发 Android 应用程序的同时应该对如下特点进行考虑。

3.3.1　有限的资源

虽然现在的智能手机的硬件配置已经越来越高,甚至单个的设备配置已经能够与一些个人计算机媲美,但是它们所拥有的资源仍然存在着一些局限性。一个最基本的并且尚未得到很好解决的限制就是移动设备的电池容量限制,移动设备的使用方式决定了其不具备能够持续不断的供电能力,然而各种硬件的运行无不是建立在对电源的依赖之上的,可以说一旦一个移动设备的电池电量消耗完毕,它就基本失去了所有的作用。处理器执行的每一

次计算、对内存的每一次刷新操作、屏幕上的每一个像素显示都需要使用电池提供的能量。

然而，受限于移动设备的体积，电池的体积往往不能够被设计得过大，这就直接导致了电源容量的有限性，而用户并不希望充电次数过于频繁(也不是任何条件下都能够顺利地为设备充电)。在这种情况下，移动设备的 CPU 频率往往被限制在相对较低的水平(通常在1GB 以下)，内存容量通常也不会太大(1GB 左右)，外存容量则通常只有几十亿字节或者最多不超过 100GB，在这些资源上的随意扩大都会导致移动设备的续航能力的大大降低，这也解释了为何目前市场上的智能手机通常只能够一次充电使用一天的情况。

Android 系统结构在进行设计时已经较为充分地考虑到了这个限制条件，因此 Android 应用程序的运行机制能够使得其功耗保持在相对较低的一个水平，然而作为一个智能手机的操作系统，在它之上势必会安装众多的应用程序，这些应用程序来自于众多的开发者，因此，作为一个自律的开发者，应当充分考虑到这一点，尽量少而优地去使用系统资源。

3.3.2 应用程序之间的复用

应用程序的复用性一直是一个广泛探讨的问题，如果应用程序是可复用的，那么它将能够在复用的过程中发挥出更多的价值。对已有程序的复用也使得我们在开发的过程中专注于对新功能的实现以及具体的整合各种组件的工作，从而极大地提高开发的效率并且能够有更多的时间去保证开发的质量。这种复用性在互联网上已经逐渐地体现出了它的价值，可以通过直接使用其他应用程序所提供的数据和方法接口，快速实现所需要的功能，一个最典型的例子就是 Google 地图以及其他类似的在线地图，在 Google 地图所提供的强大地图数据及搜索接口的帮助下，仅仅通过几行 JavaScript 代码就可以在网页上实现定制的地图功能，甚至包括清晰的卫星图像、实时更新的交通情况等。

Android 平台就是在这样的理念下构建的，不同于其他的一些移动操作系统，在 Android 中，借助于 Android 所支持的一些进程间通信的机制，可以方便地使应用程序与其他应用程序相互协作，具体的进程间通信的相关知识将在第 6 章中进行介绍。

3.3.3 可互换的应用程序

可互换的应用程序，意思就是在 Android 中任何应用程序都是可以互换的，只要应用程序声明可以处理某种类型的事件(即 Intent，在后面将会对其进行介绍)，那么它就可以成为处理这种事件的待选方案。

为什么要提到这样的一个概念呢？举例说明，在其他的一些手机操作系统上，通常只允许第三方开发的应用程序去使用系统指定的一些应用程序所提供的接口，例如在 Windows Mobile 上，当应用程序需要自动发送一封邮件时，必须在代码中显式地调用 Pocket Outlook 应用程序提供的发送邮件的接口，并且只能够通过这种方式来自动发送邮件，如果用户想使用另外的由第三方提供的邮件发送应用程序，就会在实现上显得相当困难，通常情况下必须让应用程序事先和这种应用程序建立起某种约定，这显然是不方便的。

在 Android 中，这种窘迫的情况将不复存在，Android 提供了一个基于 Intent 的机制，这个机制不依赖于任何应用程序的具体实现方式。通过 Intent 机制，当在需要发送一封电子邮件时，不需要具体指明需要通过何种应用程序来执行这个操作，只需要向操作系统发送

一个带有这个操作信息的 Intent，简单地说就是告诉操作系统你想要发送一封邮件这个意图(Intent 正是意图的意思)，随后操作系统将通过 Intent 机制查询系统中能够对这个 Intent 进行处理的应用程序，然后为用户提供一个能够处理这个 Intent 的应用程序列表，经过用户选择后完成这个发送邮件的操作。通过这种机制，可以使得需要使用某种服务的应用程序不需要了解提供该服务的应用程序的具体细节，从而降低了应用程序之间的耦合度，使得开发过程更有独立性，还可以为用户提供多种可选择的应用程序方案，从而提高用户的满意度。

3.4　应用程序结构

本节将对 Android 应用程序架构的组成部分进行介绍。一个常规的 Android 程序主要由 Activity、Service、Content Provider、Broadcast Receiver 这 4 个部分组成。但是并不是所有的 Android 应用程序都必须包含这 4 个部分，例如最简单的 Hello World 就只实现了一个 Activity，通常一个应用程序会包含一个以上的 Activity，其他组件则是根据具体的需求而进行补充。一个 Android 应用程序的完整结构如图 3-1 所示。

AndroidManifest.xml 文件相当于应用程序对外界开放的一个说明文档，外界只能够通过这个文件来对应用程序进行了解，同时它也承担了向系统注册组件的责任，只有在 AndroidManifest.xml 文件进行注册的组件才是被系统认可执行的。下面依次介绍每个组成部分。

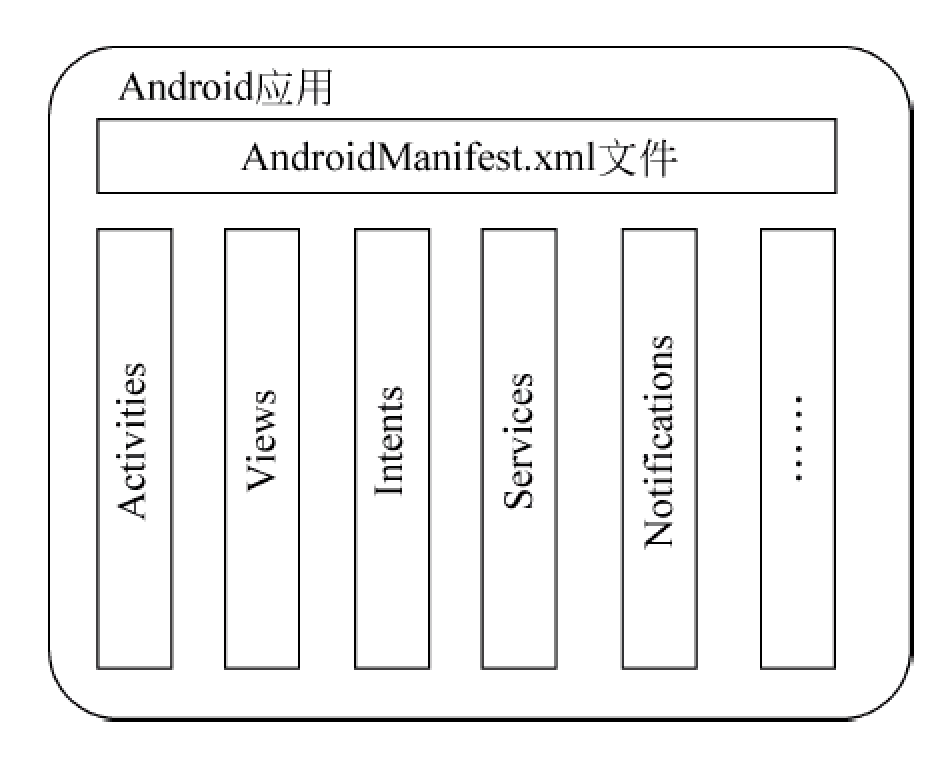


图 3-1　Android 应用程序结构

3.4.1　Activity

Activity，译为中文即活动的意思，它是 Android 中最普通的模块之一，也是开发者最常遇到的模块之一。在 Android 程序中，一个 Activity 就对应于手机屏幕的一个显示界面，类似于浏览器中的一个网页。通常我们会在 Activity 中添加一些 UI 组件(第 4 章将会对其进行介绍)，并对这些组件实现相应的事件处理。一个 Android 应用程序中可能涉及多个 Activity，并且根据需要能够在这几个 Activity 中进行跳转。打开一个新的 Activity 时会将当前的 Activity 置为暂停状态并压入堆栈，Android 默认会将每个应用从开始到当前的每个 Activity 都保存到堆栈中，也可以通过在代码中进行相关的设置使得一些无须保留的 Activity 不被系统压入堆栈保存。

在应用程序中每一个 Activity 都拥有自己的生命周期，这个生命周期由系统来实现统一的管理。一个 Activity 有 3 个基本的状态，即活动状态(running)、暂停状态(paused)以及停止状态(stopped)。这 3 个状态之间的转换方式如下：

- 当其在前台运行时(即在 Activity 当前任务的堆栈顶)，即为活动状态(运行状态)。这时 Activity 会响应用户的操作。
- 当 Activity 失去焦点但是对用户仍然可见时为 paused 状态。此时，其他的 Activity 存在于自己之上，这种情况可能是透明或者被非全部覆盖(如非全屏的对话框)。所

以其中一些处于暂停状态的 Activity 也可以被显示。一个暂停的 Activity 仍然是处于活动状态的(它维护着所有的状态保存着信息,并且依然附着在窗口管理器)。

- 如果一个 Activity 完全被另一个 Activity 所掩盖,那它的状态会变为 stopped。此时仍然保存着状态信息。
- 当其他应用程序需要使用更多的内存时,系统有可能会终止处于 paused 状态或 stopped 状态的 Activity(系统会在终止 Activity 之前对状态进行保存)。当其再次需要显示时,系统会重新运行该 Activity 并且加载所保存的状态信息。

一个 Activity 从开始执行到最后终止退出内存,将会经过如下一个完整的生命周期,这个生命周期由系统来统一维护,开发者在实现一个 Activity 时需要正确地把握好这个完整的生命周期,以使得 Activity 可以正常工作,最重要的一项工作通常是在 Activity 进入暂停状态或停止状态时需要对 Activity 的当前运行状态进行保存,以便于 Activity 重新进入运行状态时能够根据保存的信息正常地进行恢复,否则可能使应用程序出现意想不到的运行结果,甚至出现丢失数据的危险状况。当然,Activity 通常是在应用程序中扮演着视图的角色,通常可以将一些需要一直运行的逻辑作为 Service 放在后台运行,从而避免过于烦琐的状态保存和恢复工作。

如图 3-2 所示是描述 Activity 生命周期的框图,该图摘自 Android SDK 文档。

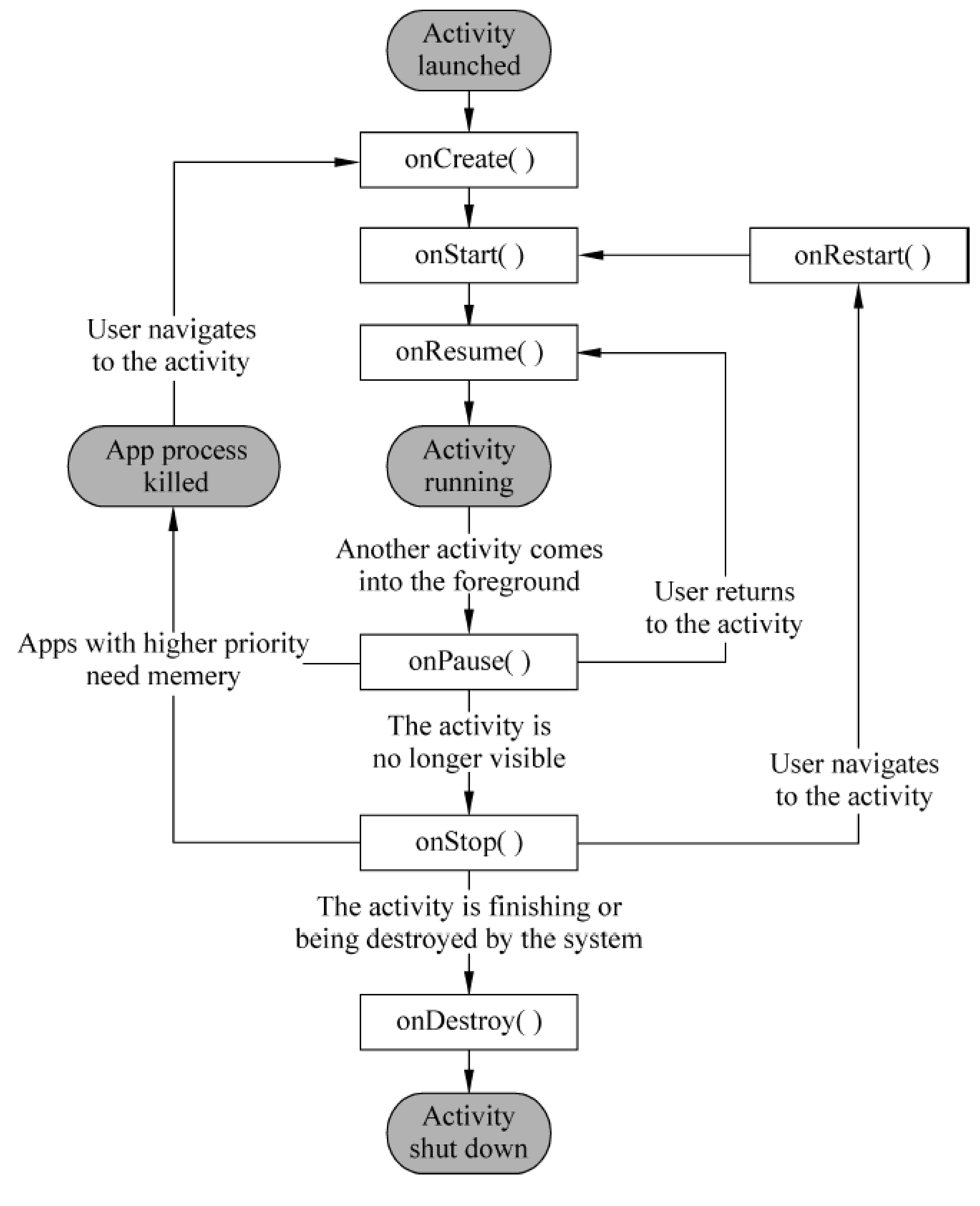


图 3-2 Activity 生命周期

图 3-2 详细表示了 Activity 生命周期，其中包括了 3 个主要的循环结构，其中每一个较小的循环都是较大循环的子集，这 3 个循环结构由大至小分别为：

- 完整的 Activity 生命周期。这个周期循环从该 Activity 的 onCreate()方法第一次被调用开始，直到 onDestroy()方法被调用结束。在 onCreate()方法中，Activity 会对所有的全局状态进行初始化，并在 onDestroy()方法中释放所有资源。
- Activity 的可见生命周期。这个周期从 onStart()方法被调用时开始，直到 onStop()方法被调用时结束，在这个周期中 Activity 对于用户是可见的，但是也有可能不处于 Activity 栈的最上方即不是可交互的。在这个周期中可以获取资源并对 UI 进行更新。
- Activity 前台生命周期。在这个周期中 Activity 始终处于栈的顶端并且可以与用户交互。周期从 onResume()方法被调用时开始直到 onPause()方法被调用时结束，对于一个 Activity 来说，这两个方法会被十分频繁地调用到，例如当 Android 进入休眠状态或者该 Activity 调用了新的 Activity。

Activity 的生命周期涉及如上一些方法，正如 Activity 类的源码实现框架：

```
public class Activity extends ApplicationContext {
      protected void onCreate(Bundle savedInstanceState);
      protected void onStart();
      protected void onRestart();
      protected void onResume();
      protected void onPause();
      protected void onStop();
      protected void onDestroy();
}
```

开发者可以通过重写这些方法，从而在 Activity 生命周期的适当位置对 Activity 进行需要的管理和操作。例如，通常在 onCreate()方法中对 Activity 进行初始化操作，在 onPause()方法中对 Activity 状态进行保存，在 onResume()方法中恢复 Activity 状态等。这里顺便对 Android 的应用程序的生命周期进行一下说明。在多数情况下，所有的 Android 应用程序运行在它们各自的 Linux 进程中。应用程序进程会在需要被运行时被创建，一般到运行结束才释放内存。但是当系统内存资源不足时，系统可以回收内存并分配给其他需要内存的程序。

Android 的一个非常重要的特点就在于它对生命周期的控制。Android 应用程序生命周期的控制并不是直接由应用程序本身来完成的，而是由系统通过与应用程序联合来进行控制的。这样系统就可以知道哪些应用程序正在运行，哪些对用户来说更加重要，以及目前系统的可用内存是多大等信息。

对于开发者而言，理解各种不同的应用程序组件(特别是 Activity、Service 和 IntentReceiver)及它们在应用程序进程的生命周期中所起到的作用是十分重要的。一个使用不当的应用组件会导致系统 kill 掉该应用的进程。

在内存不足时由系统决定哪些进程被 kill 掉，Android 的方法就是将所有进程放入一个基于组件的运行与状态的“重要性层级”中。以下是重要性的排序：

- foreground process(显著进程)持有一个与用户交互的屏幕顶层的 Activity，或者一

个目前正在运行的 IntentReceiver。系统中这样的进程并不多,一般情况下仅在内存已被耗尽,且不足以维持进程运行时万不得已而被 kill 掉。通常这个时候设备已经到了存储器页面调整状态,因此这时 kill 是为了保证用户界面的响应而不得已要做的(防止假死)。

- visible process(可见进程)持有一个用户在屏幕上可见的但并不是最显著位置的 Activity(onPause() 方法被调用)。例如,如果新的显著进程(对话框)被显示并允许之前的 Activity 显示为背景,这样的进程被认为相当重要而不可以终止,除非是为了保证显著进程可以运行而进行的终止操作。
- service process(服务进程)持有一个通过 startService() 方法启动的 Service,尽管这些进程并不会被用户直接看到(Service 不提供界面显示),但它同样在处理一些用户很关心的事情,例如在后台的播放的音乐以及文件的上传下载等操作。所以系统会一直维持这些进程的运行,除非内存已无法维持显著进程与可见进程的运行。
- background process(后台进程) 持有一个用户已不可见的 Activity(onStop() 方法被调用)。这些进程在内存中的存在或消失对用户来说没有直接的影响。系统可以在任意时刻 kill 掉这类进程并释放内存给前面 3 类进程。通常系统中这类进程会很多,它们会被保存在一个 LRU 列表中,即在系统内存不足时,用户最近最少访问的进程将最先被 kill 掉。
- empty process(空进程)不包含任何应用程序组件。保留这些进程是为了充当缓存的作用,以提高应用程序下一次启动的速度,因此系统会优先使用这些进程所持有的资源。kill 空进程的操作通常是为了平衡系统资源在内核缓存和进程缓存之间的分配问题。

3.4.2 Service

Service 即"服务",它与 Activity 属于同一等级的应用程序组件,不同的是 Activity 拥有前台运行的用户界面,而 Service 不拥有界面,通常 Service 需要通过某个 Activity 或者其他 Context 对象来调用。Service 在后台运行,它不能与用户直接进行交互。在默认情况下,Service 运行在应用程序进程的主线程之中,但如果需要在 Service 中处理一些诸如连接网络等耗时操作时,就应该将其放在单独的线程中进行处理,避免阻塞用户界面。可以通过 Context. startService() 和 Context. bindService() 两种方式来启动 Service。与 Activity 一样,Service 也拥有自己的生命周期,由于其不存在界面,因此相对于 Activity 的生命周期来说比较简单。

1. 使用 Context.startService() 方法启动 Service

该种方法启动的 Service 在运行中会经历如下的过程:Context. startService→onCreate()→onStart()→Service running→onDestroy()→Service 终止。如果 Service 处于未运行的状态,则需要先调用 onCreate() 然后再调用 onStart() 的顺序来启动;如果 Service 已经处于运行状态,则只需要调用 onStart() 来启动 Service 即可。onStart() 方法可被重复调用多次,即可以多次调用 startService() 方法,Service 将会接收这些请求然后通过 onStartCommand() 方法

来依次对这些请求做出处理。如果是使用这种方式启动 Service,那么关闭 Service 的方法就很简单,通过 stopService()方法直接停止 Service,Service 内部则会通过 stopSelf()方法来确保在退出 Service 之前已经完成了对所有 Intent 的处理,再调用 onDestroy()方法销毁 Service。如果调用者直接退出而没有调用 stopService,那么 Service 会在后台一直运行,直到该 Service 的调用者再次启动后通过 stopService 关闭 Service。这种调用方式的 Service 生命周期:onCreate()→onStart()(多次)→onDestroy()。

对于已开始的 Service 来说,有两种不同的执行模式,这取决于 onStartCommand()方法的返回值,如果返回 Service. START_STICKY,那么这个 Service 就仅在显式地调用 stop 方法时才会终止;如果返回 START_NOT_STICKY 或者 START_REDELIVER_INTENT,那么 Service 将仅仅会在执行命令的时候才会保持在运行状态。

2. 使用 Context. bindService()方法启动 Service

该种方法启动 Service 需要经历如下步骤:Context. bindService()→onCreate()→onBind()→Service running→stopService()→onUnbind()→onDestroy()→Service stop。onBind 将返回给客户端一个 IBinder 接口实例,这个实例允许客户端回调服务方法,这个 IBinder 对象通常是通过 aidl 接口定义文件定义的。这种方法会把调用者(Context、Activity 等)和 Service 绑定在一起,并且 Service 在这个连接建立的过程中一直保持运行状态。当 Context 退出时,Service 会调用 onUnbind()→onDestroy()退出。所以这种调用方式下 Service 的生命周期为:onCreate()→onBind()(与第一种方式不同,这里 onBind()只能绑定一次,不可多次绑定)→onUnbind()→onDestroy()。

与 Activity 类似,创建一个 Service 的基本方法是通过继承 android. app. Service 类来实现 Service 自己的服务。另外也需要在 AndroidManifest. xml 中注册 Service。在 Service 的生命周期中,只有 onStart()方法可被多次调用,其他的 onCreate()、onBind()、onUnbind()、onDestroy()等在一个生命周期中,都只能被调用一次。

使用 Service 的应用场景有很多,例如播放音视频文件时,用户启动了其他的 Activity,这时候播放程序还会在后台继续播放;又如需要检测 SD 卡上文件变化的应用程序、需要在后台记录用户的地理信息位置改变信息的应用程序等。因为 Service 是后台运行的,所以它总是在程序需要后台服务的时候调用。以音乐播放器的应用为例,在播放器应用运行的过程中会涉及很多个 Activity,用户可以在不同的 Activity 下进行不同的操作而达到不同的目的,如用户进入歌曲菜单选择歌曲进行播放,这时就需要借助 Service 后台运行的特点,使用户在导航到其他 Activity 时音乐仍继续播放。在此应用中,音乐播放 Activity 会使用 Context. startService()启动一个 Service,在后台开始播放音乐,系统会一直保持这个 Service 直到音乐播放结束,在播放过程中还可以进行暂停、继续等操作。

3.4.3 Content Provider

Content Provider,即内容提供者,它的用途是为应用程序的数据提供存储和检索,并且使得这一部分数据能够被其他所有的应用程序所访问。ContentProvider 是 Android 中跨应用程序共享数据的唯一途径,因为 Android 并没有直接提供能够被所有应用程序访问的共享区域。

随 Android 系统一同发布的组件中就包含了对一些常用数据类型进行访问的 Content Provider(例如音频、视频、图片、联系人信息等),在应用程序中可以通过这些数据的 ContentProvider 来实现对它们的检索和访问,前提条件是有它们的访问许可(请参见 3.4.7 节)。

有关 ContentProvider 的更多介绍,请参见 5.3 节。

3.4.4 Intent

1. Intent 的作用

Intent(意图),通常配合 IntentFilter(意图过滤器)使用,Android 应用程序的 3 大核心组件——Activity、Service 和 BroadcastReceiver(请参见 3.4.5 节)之间通过 Intent 消息机制来交互。Intent 是一个将要执行的动作的抽象描述,一般来说是作为参数来使用,由 Intent 来协助完成 Android 各个组件之间的通信。比如说调用 startActivity()来启动一个 Activity,或者由 BroadcastIntent()来传递给所有感兴趣的 BroadcastReceiver,再或者由 startService()/bindservice()来启动一个后台的 service。可以看出,Intent 主要是用来启动其他的 Activity 或者 Service,所以可以将 Intent 理解成各种应用程序组件之间的黏合剂。

2. Intent 的构成

要在不同的 Activity 或其他组件之间传递数据,就要在 Intent 中包含相应的信息和数据,一般地,Intent 应该包括如下两个最基本的数据:

- Action——用来指明要实施的动作是什么,比如说 ACTION_VIEW(浏览网页)、ACTION_EDIT(编辑文本)等。具体可以查阅 Android SDK→reference 中的 android.content.intent 类,其中的常量定义了所有的 Action 类型。
- Data——要操作的具体数据,一般由一个 Uri 来确定。

通过 Action 和 Data 的内容基本就能确定你的意图(Intent)到底是需要进行什么样的操作,下面给出两个简单 Action 与 Data 共同确定一个操作意图的例子:

```
ACTION_VIEW  content://contacts/1  //显示 identifier 为 1 的联系人的信息
ACTION_DIAL  content://contacts/1  //给这个联系人打电话
```

除了 Action 和 data 这两个最基本的元素外,Intent 还包括一些其他的元素:

- Category(类别):这个选项指定了将要执行的这个 action 的其他一些额外的信息,例如 LAUNCHER_CATEGORY 表示 Intent 的接受者应该在 Launcher 中作为顶级应用出现;而 ALTERNATIVE_CATEGORY 表示当前的 Intent 是一系列的可选动作中的一个,这些动作可以在同一块数据上执行。具体同样可以参考 android SDK→reference 中的 Android.content.intent 类。
- Type(数据类型):显式指定 Intent 的数据类型(MIME)。一般 Intent 的数据类型能够根据数据本身进行判定,但是通过设置这个属性,可以强制采用显式指定的类型而不再进行推导。
- Component(组件):指定 Intent 的目标组件的类名称。通常 Android 会根据 Intent 中包含的其他属性的信息,比如 action、data/type、category 进行查找,最终找到一个与之匹配的目标组件。但是,如果 Component 属性有指定的组件,将直接使用它

指定的组件，而不再执行上述查找过程。指定了这个属性以后，Intent 的其他所有属性都是可选的。

- Extras(附加信息)，是其他所有附加信息的集合。使用 Extras 可以为组件提供扩展信息，比如，如果要执行"发送电子邮件"这个动作，可以将电子邮件的标题、正文等保存在 Extras 中，传给电子邮件发送组件。

3. Intent 解析方法

Intent 是由应用程序向系统发出的一个意图请求，表示需要完成某种任务，然后由系统来确定将这个 Intent 交给哪一个应用程序去处理，因此，如果某个应用程序能够处理某种类型的 Intent，就应当向系统进行声明，以便于系统能够正常地对 Intent 进行调度。

Intent 可以分为两大类：显式 Intent(Explicit Intents)和 Intent Filter，即隐式 Intent(Implicit Intents)，显式 Intent 即需要通过显式地指定目标组件的名称，因此一般来说显式 Intent 主要用于应用程序内部的组件之间传递消息，而隐式 Intent 不需要指定目标组件的名称，因此用于多个应用程序之间的交互。Intent Filter 进行匹配时的三要素是 Intent 的动作(Action)、数据(Data)以及类别(Type)。实际上，一个隐式 Intent 请求要能够传递给目标组件，必须通过这 3 个方面的检查。如果任何一方面不匹配，Android 都不会将该隐式 Intent 传递给目标组件。应用程序组件为了告诉 Android 自己能响应或处理哪些隐式 Intent 请求，可以声明一个甚至多个 Intent Filter。每个 Intent Filter 描述该组件所能响应 Intent 请求的能力——组件希望接收什么类型的请求行为、什么类型的请求数据。比如在需要请求网页浏览器时，网页浏览器程序的 Intent Filter 就应该声明它所希望接收的 Intent Action 是 WEB_SEARCH_ACTION，以及与之相关的请求数据是网页地址 URI 格式。为组件声明自己的 Intent Filter 最常见的方法就是在 AndroidManifest. xml 文件中用属性<intent-filter>描述组件的 Intent Filter。

接下来说明一下动作、数据(Data)以及类别的匹配规则。

1) Action 匹配

<intent-filter>元素中可以包括子元素<action>，例如：

```
<intent-filter>
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
</intent-filter>
```

一个<intent-filter>节点下至少应该包含一个<action>子节点，否则任何 Intent 请求都不能和该<intent-filter>匹配。如果 Intent 请求的 Action 和< intent-filter>中某一条< action>匹配，那么该 Intent 就通过了这条<intent-filter>的匹配。如果 Intent 请求或<intent-filter>两者中有任意一方没有说明具体的 Action 类型，那么会出现下面两种情况。

- 如果< intent-filter>中没有包含任何 Action 类型，那么无论什么 Intent 请求都无法和这条< intent-filter>匹配。
- 反之，如果 Intent 请求中没有设定 Action 类型，那么只要<intent-filter>中包含有

Action 类型，这个 Intent 请求能够匹配该<intent-filter>。

2）类别匹配

<intent-filter>元素可以包含< category>子元素，例如：

```
<intent-filter>
    <category android:name="android.Intent.Category.DEFAULT" />
    <category android:name="android.Intent.Category.BROWSABLE" />
</intent-filter>
```

当 Intent 请求中所有的 Category 与组件中某一个 IntentFilter 的<category>完全匹配时，该 Intent 请求将成功匹配，IntentFilter 中其他的<category>声明并不会导致匹配失败。一个没有指定任何类别的 IntentFilter 只匹配没有设置类别的 Intent 请求。

3）数据匹配

数据匹配在<intent-filter>中的描述如下：

```
<intent-filter>
    <data
        android:scheme="http"
        android:type="video/mpeg" />
    <data
        android:scheme="http"
        android:type="audio/mpeg" />
</intent-filter>
```

通过如上方式指定了应用程序能够接受的 Intent 请求的数据 URI 格式和数据类型，URI 被分成 3 部分来进行匹配：scheme、authority 和 path。其中，用 setData()设定的 Intent 请求的 URI 数据类型和 scheme 必须与 IntentFilter 中所指定的一致。若 IntentFilter 中还指定了 authority 或 path，只有全部都匹配才会成功匹配。

4. Intent 示例

创建一个 Intent 用于启动拨号程序，并且为拨号程序设置一个默认号码：

```
Intent i = new Intent(Intent.ACTION_DIAL,Uri.
parse("tel://13800138000"));
startActivity(i);
```

将会得到如图 3-3 所示的效果。



图 3-3 拨号 Intent

3.4.5 BroadcastReceiver

BroadcastReceiver，即“广播接收者”，它用于异步接收广播 Intent，广播 Intent 的发送是通过

Context. sendBroadcast()方法、Context. sendOrderedBroadcast()方法或者 Context. sendStickyBroadcast()方法来进行的。通常一个广播 Intent 可以被订阅了此 Intent 的多个广播接收者所接收,广播接收者和 JMS 中的 Topic 消息接收者很相似。要实现一个广播接收者方法如下(以接收"新接收到短信"广播为例):

- 继承 BroadcastReceiver,并重写 onReceive()方法。

```
public class IncomingSMSReceiver extends BroadcastReceiver {
 @Override public void onReceive(Context context, Intent intent) {
 }
 }
```

- 订阅感兴趣的广播 Intent(即"新接收到短信"),订阅方法有两种。

第一种方法:在代码中订阅。

```
IntentFilter filter = new IntentFilter("android.provider.Telephony.SMS_RECEIVED");
IncomingSMSReceiver receiver = new IncomingSMSReceiver();
 registerReceiver(receiver, filter);
```

第二种方法:在 AndroidManifest. xml 中的<application>节点中进行订阅。

```
<receiver android:name=".IncomingSMSReceiver">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

通常一个 BroadcastReceiver 对象的生命周期不会超过 5 秒,所以在 BroadcastReceiver 中不能进行比较耗时的操作。如果需要完成一项比较耗时的工作,可以通过发送 Intent 给 Activity 或 Service,由 Activity 或 Service 来完成。如下面的代码所示:

```
public class IncomingSMSReceiver extends BroadcastReceiver {
    @Override public void onReceive(Context context, Intent intent) {
        //完成一些比较简单的操作
        doSomethingHere();
        //由服务来完成比较耗时的操作
        Intent service = new Intent(context, YourServiceHere.class);
        context.startService(service);
        //由 Activity 来完成比较耗时的操作
        Intent newIntent = new Intent(context, YourActivityHere.class);
        context.startActivity(newIntent);
 }
}
```

3.4.6 应用程序资源

在应用程序中除了代码之外,通常需要使用到其他一些非代码的资源,例如图片、字符串等,为了提高代码的灵活性,应该将这些资源作为外部的文件进行存放,而不是将它们硬

编码到代码中，这样就能够在需要改变外部资源时不必去对代码进行改变。在 Android 中，将这些资源存放在外部还有另外一个好处，那就是可以为应用程序提供分别适用于多种设备环境的资源，使应用程序能够在不同分辨率、不同像素密度以及不同的国家和地区都能够很好地工作，常见的例如为不同国家和地区提供正确的界面显示语言、为不同分辨率的设备提供不同分辨率的图标以及在屏幕横向放置和纵向放置时采用不同的布局界面等。下面就简要介绍一下这种机制的具体实现方法，然后再介绍一下如何在代码中使用这些外部资源。

1. 为应用程序提供资源

应用程序需要使用的一些资源通常存放在 res/文件夹下，例如现在版本的 ADT 在新建一个项目时，会默认得到如图 3-4 所示的一个 res/结构。

```
res
    drawable-hdpi
    drawable-ldpi
    drawable-mdpi
    layout
    values
```

图 3-4 res 文件夹结构

如图 3-4 所示，默认的 res/文件夹中存在 3 种类型的文件夹，它们分别用于存放对应的图片资源文件、界面布局文件以及其他的一些数值的文件。实际上，这里面已经给出了一个用于为不同配置的设备环境应用不同资源文件的例子，即 3 种带有不同后缀名的 drawable 文件夹，它们分别对应 3 种不同的屏幕类型：hdpi（高像素密度）、ldpi（低像素密度）和 mdpi（中等像素密度）。这样，当应用程序运行时，它会根据目前设备的属性来应用这 3 种资源中最适合的一种，通常这 3 个文件夹下必须包含有相同名称的资源文件，否则可能会导致运行出错。

Android 正是通过这样的命名规则来决定每一个文件夹内的资源是否能够工作在对应的设备上，因此这些文件夹名称是严格规定的，文件夹名称的含义如下：

```
<resources_name>-<config_qualifier>
```

其中第一项 resource_name 代表的是资源的类型，后面的 config_qualifer 代表了其使用的类型参数，它们之间必须使用减号“－”进行分隔，一个文件夹名称可以包含若干个 config_qualifer，从而能够更加精确地进行设备类型匹配，例如 drawable-en-port 就代表了这个图像资源适用于 en（英文）和竖直屏幕（port）的设备。

2. 使用资源

在应用程序中如果要使用 res/中的资源，需要遵从 Android 的约定，在代码中需要根据各个资源的 ID 来进行引用，这个资源 ID 是由 aapt 工具（由 Android SDK 提供）在 Android 项目进行编译时自动为项目生成的，也就是在项目树下面通常可以看见的 gen/文件夹下的 R.java 文件。aapt 能够生成一系列具名常量用于表示相应的资源，使得在代码中可以通过如下的形式进行引用（以图片资源为例）：

```
R.drawable.icon
```

其中 R 是固定的前缀，代表了对 R.java 文件的引用，第二部分则是对应于各种资源的类型（如 drawable、layout、string 等，较为完整的列表将在后面给出），最后一部分则是对应了具体的资源名称，类似于图片资源这种以文件形式提供的资源，这个资源名称就是其文件名（不包括后缀），而对于字符串这种简单数据则是使用其在具体文件中声明的名称（通过

android:name 属性声明)。

如果需要在 xml 文件中使用,则需要通过如下的方式:

```
@drawable/icon
```

具体的规则与前面提到的完全一致,只是"R."变成了"@",而分隔符由"."变为了"/"。

另外需要注意的是,Android 本身提供了一些常用的标准资源,例如列表的布局,可以通过如下的形式来引用:

```
android.R.layout.simple_list_item_1
```

3. 常用的资源类型

在 res/通常会用到如下资源类型:

- res/anim/——实现一些动画所需要的资源,通过 R.anim 的形式引用。
- res/color/——一些颜色资源,用于为某个视图的不同状态提供不同的颜色(例如按钮被单击和被按下采用不同颜色),通过 R.color 形式引用。
- res/drawable/——图片资源,通过 R.drawable 形式引用。
- res/layout/——布局文件资源,通过 R.layout 形式引用。
- res/menu/——目录项资源,为某个目录提供内容,通过 R.menu 形式引用。
- res/raw/——存放一些原始数据,例如视频文件或者音频文件,通常使用 openRawResource()方法来获取文件的输入流。
- res/values/——这个文件加下面用于存放一些相对简单的数据,例如字符串、整型数据、布尔数据、尺寸数据以及它们的数组,还包括复数等,或者用于定义某种风格的 style 数据,例如定义某种字体或某种界面主题。相应的引用形式分别有 R.string、R.array、R.plurals、R.bool、R.integer、R.dimen、R.id、R.style 等。

需要注意的一点是,如果需要对一些外部文件进行访问并且修改它们,那么存放在 res/文件夹下并不能够满足这样的需求,因为访问 res/下的文件只能够通过前面所述的只读方式打开,这时就需要将所要存取的这类文件存放在 assets/文件夹下,然后借助 AssetManager 对象来获取对这些文件的访问。

3.4.7 安全与权限机制

Android 是一个权限被严格分隔的操作系统,Android 使用了 Linux 内核,这使得每一个应用程序都运行在一个截然不同的系统环境下,每一个应用程序都具有一个 Linux 用户 ID 和组 ID,这也使得系统的各个组成部分都运行在相互分离的环境下。Linux 将所有的应用程序以及系统组件都分隔开来,这样能够获得较高的安全性,但是同时也使得应用程序之间不能以直接的方式相互访问。

为此,Android 提供了一个以 permission 为基础的权限机制,这种机制可以使得 Android 的应用组件之间能够突破一般的进程之间相互不能访问的限制而实现一些特定的相互访问操作。借助名为"per-URI"的机制,使得应用程序能够临时性地获取到某些数据(通过 URI 指定)的访问权限,这种机制通常运用在 ContentProvider 之上,使得

ContentProvider 能够支持应用程序这种临时权限的申请，支持这个机制的前提是在 ContentProvider 中通过 android:grantUriPermissions 属性或者＜grant-uri-permissions＞标签实现该功能，具体的方法将在 5.3 节进行介绍。

Android 权限机制的设计核心是：在默认的情况下，任何应用程序都没有权限去对其他应用程序、操作系统和用户进行可能造成任何的负面影响的操作，这些操作包括但不限于读写用户的隐私数据（比如联系人和邮件）、读写其他应用程序的文件、进行网络访问、使设备保持唤醒状态等操作。

在 Android 中，每一个应用程序都运行在一个独立的沙盒中，如果应用程序想要与外界共享自己的资源和数据，那么必须显式地进行说明（通过 AndroidManifest.xml 文件）。当用户在安装应用程序时，系统会向用户申请相应的使用权限。一个应用程序的权限是以静态的方式在 AndroidManifest.xml 进行声明的，而不能够在运行时实时地申请权限，因为这会降低用户体验。

在应用程序进行安装时，Android 就会为这个应用程序分配一个在当前系统中唯一的 Linux 用户 ID，并且在这个应用程序的生命周期内（安装→卸载），这个用户 ID 都是保持不变的状态。因此，两个应用程序通常不能够运行在同一个进程中，因为它们都是以不同的 Linux 用户身份来运行的。不过，可以通过在 AndroidManifest.xml 文件的 manifest 标签的 sharedUserId 属性来使得不同的应用程序可以被赋予相同的 Linux 用户 ID，在这种情况下，这两个应用程序通常会被视为一个大的应用程序，它们拥有相同的用户 ID 和文件访问权限。

每一个应用程序所存储的数据和文件同样会被分配它所属的应用程序的用户 ID，正因为如此，一个应用程序通常不能够去访问另一个应用程序的文件。但是可以通过一些特殊的文件创建方式，使得某些文件能够被外部应用程序读写，如通过 openFileOutput()方法创建文件时，使用 MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE 标志来使得这个文件可被其他的应用程序读写。

任何权限的申请都需要通过 AndroidManifest.xml 文件来完成，有关 AndroidManifest.xml 的更多信息，请参见 3.4.8 节的内容。

3.4.8 AndroidManifest.xml

AndroidManifest.xml 是每个 Android 项目中必须定义的文件。它位于项目的根目录，描述了 package 中的全局数据，包括 package 中的组件（如 Activities、Services 等）以及这些组件各自的实现类，还有各种能被处理的数据以及其他属性。

除了声明程序中的 Activities、Content Providers、Services 和 Intent Receivers 组件之外，permissions 和 instrumentation（安全控制和测试）也需要在文件中进行声明。

下面是一个简单的 AndroidManifest.xml 文件示例，取自于 HelloWorld：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.android.HelloWorld"  android:versionCode="1"
        android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
```

```
        <activity android:name = ".HelloWorld"
                  android:label = "@string/app_name">
            <intent - filter>
                <action android:name = "android.intent.action.MAIN" />
                <category android:name = "android.intent.category.LAUNCHER" />
            </intent - filter>
        </activity>
    </application>
</manifest>
```

其中需要注意的有以下几点：

(1) 通常 manifests 节点下包含一个<application>节点，<application>内定义了所有的应用程序组件及属性。

(2) 任何被用户看做是顶层应用程序的应用程序并且需要能被程序启动器所启动的 package，需要包含至少一个 Activity 组件来响应 MAIN 操作，并且归属于 LAUNCHER 类，如上述代码中所见。

下面介绍一下 AndroidMainfest.xml 中需要使用到的部分标签的含义：

• manifest。

根节点，描述 package 的所有信息。后面介绍的标签都放在 mainfest 节点下。即 AndroidMainfest.xml 是以 manifest 标签开头和结尾的。

• uses-permission。

请求 package 正常工作所需被赋予的安全许可。该节点可出现 0 或 $n(n \geqslant 1)$次。

• permission。

声明安全许可来限制其他程序访问 package 中的组件和功能，这种许可权限是指自身 package 对外部发放的权限。该节点可出现 0 或 $n(n \geqslant 1)$次。

• instrumentation。

声明用于测试此 package 的其他类或包。该节点可出现 0 或 $n(n \geqslant 1)$次。

• application。

包含 package 中 application 级别组件声明的节点。此节点也可包含 application 中全局和默认的属性，如标签、图标、主题等。该节点只允许出现 0 或 1 次。在它之下可放置的标签有 activity、service 等。

• activity。

Activity 是用来与用户交互的主要工具。当用户打开一个应用程序的初始页面就是一个 Activity 时，大部分被使用到的其他页面也由不同的 Activity 实现，所有需要使用到的 Activity 都必须声明在各自的 activity 标签中。

注意：每一个 activity 必须对应一个<activity>标签，无论它给外部使用或是只用于自己的 package 中。如果一个 activity 没有用标签声明，那么它将不能被运行。另外，为了在程序运行的时候能够查找到 activity，每个 activity 标签可能会包含一个或多个<intent-filter>元素来描述该 activity 所支持的操作。

• intent-filter。

该节点声明了所属组件所支持的 Intent 类型。通常在此标签下会包含随后的 action

(至少包含一个)、category、data 类型等标签。

- action——组件支持的 action 类型。
- category——组件支持的 category 类型。
- type——组件支持的 Intentdata MIME type。
- schema——组件支持的 Intentdata URI scheme。
- authority——组件支持的 Intentdata URI authority。
- path——组件支持的 Intentdata URI path。
- receiver。

IntentReceiver 使得 application 能够获知数据的改变和发生的操作,并且可以允许 application 当前不在运行状态。通常配合<intent-filter>使用。

- service。

Service 是能在后台运行的组件。通常配合<intent-filter>使用。

- provider。

ContentProvider 是用来管理持久化数据并发布给其他应用程序使用的组件。

3.5 前置技能

要进行 Android 应用开发,需要对如下技能有一些基本的了解,并且能够正确地搭建好 Android 的应用开发环境。

1. Eclipse

Android 应用开发的集成开发环境,使用 Eclipse 有助于快速、高效地进行 Android 应用开发,配合 Eclipse 强大的插件功能,几乎能够使用任何语言进行任何类型的开发工作,最重要的是,它还是可以免费获得并使用的。为此,需要对 Eclipse 的一些基本操作和快捷键进行了解,可访问 Eclipse 的官方网址:http://www.eclipse.org/。

2. Java

由于 Android 应用程序都是使用 Java 语言进行开发的,因此必须对 Java 编程语言有所了解,其基本的语法、数据结构以及面向对象的编程思想都是应该事先了解的知识。可以在如下网址获取到 Java(包括 JDK 和 JRE):http://java.com。

3. xml

Android 中的布局文件、资源文件都是采用 xml 来进行定义和配置的,因此需要对 xml 的使用方式有一个简单的了解,可以在如下网站学习 xml:http://www.w3schools.com/xml/。

4. SQLite

应用程序多多少少都需要与数据打交道,而管理数据库的最佳方式就是使用数据库,Android 应用程序通常使用 SQLite 作为数据库,无论使用何种数据库技术,都需要对数据

库的一些基本语法进行了解，比如怎样增加和删除数据，怎样正确地检索出所需要的数据等，可以在 SQlite 的官方网站上对 SQLite 进行了解和学习：http://www.sqlite.org/。

参考文献

1. 维基百科“软件开发工具包”词条：http://zh.wikipedia.org/wiki/软件开发工具包.
2. Rick Rogers / John Lombardo，Android Application Development，1st Edition，O'Reilly Media，Inc. 2009-05-26.
3. Android 官方文档：http://developer.android.com/guide/topics/fundamentals.html.

第4章 用户界面

应用程序的用户界面属于用户接口的一种，即 User Interface(UI)，它是系统和用户之间进行信息交换和操作交互的媒介，它的主要作用是实现信息内部形式与人类可接受形式之间的转换。它的目的在于使用户能够方便地并且有效率地去操作硬件以达成双向交互，从而完成所希望借助硬件去完成的工作。用户界面定义广泛，包含了人机交互（例如键盘、鼠标）与图形用户界面，可以说凡是涉及人类与机械的信息交流的领域都存在着用户界面。用户界面设计包括了对软件的人机交互、操作逻辑、界面美观的整体设计。好的 UI 不仅可以让软件变得个性有品味，更重要的是可以让软件的操作变得舒适、简单而自由，并且充分地体现软件的定位和特点。

用户界面对于应用程序的重要性，好比着装于人的重要性，俗话说："人靠衣装"，对人来说，好的着装不仅可以使得自己大方得体有精神，还可以满足功能性的需求，例如保暖、透气甚至是口袋易用性。从广义上来说，生活中所接触到的各种产品都涉及 UI 设计，例如汽车、冰箱、空调等，一个好的 UI 设计都会体现出这样的共同点：它们方便使用，易于学会，提示信息清晰明了，很少有带有歧义性的操作出现。

为了实现良好的应用程序用户界面，Android 提供了一系列用户界面组件，包括了用于对应用程序界面进行布局的布局组件、一些基本的操作控件以及消息通知组件等，用户界面的设计需要读者在实践中逐渐积累经验，因此本章仅简单地对这些知识点进行介绍，主要目的是让读者对 Android 提供的一些用户界面组件有一个较为全面的认识。

在本章开始之前，先介绍几个在用户界面设计中需要弄清楚的小常识。

在 Android 中，描述视图大小的单位通常有如下几种：

- px，表示像素（pixels）。
- dip(dp)，表示不依赖于设备的像素（device independent pixels）。
- sp，比例化的像素（scaled pixels——适用于表示字体大小）。
- pt，表示点（points）。
- in，表示英尺（inches）。
- mm，表示毫米（millimeters）。

另外，需要分清楚 margin 和 padding 这两种属性的区别：

- margin——两个独立视图之间的间距。
- padding——填充于两个视图（两个视图具有包含与被包含的关系）之间的间距。

4.1 布局类型

一个 Android 应用程序的用户界面是由若干个 View(视图)和 ViewGroup(视图群组)所组成的。从类结构上来说,View 和 ViewGroup 都属于 android.view 包,而 View 和 ViewGroup 又衍生了很多的子类。ViewGroup 是可以嵌套的,即一个 ViewGroup 中可以包含另一个或多个 ViewGroup,各种各样的 ViewGroup 和 View 就可以组成所需要的用户界面,可以用一个树形结构来描述用户界面的结构,如图 4-1 所示,一般来说,ViewGroup 是树形结构中的父节点,而 View 是属性结构中的叶节点。

通常 ViewGroup 就是那些复合型的视图控件,它们通常包含了若干个子视图控件,例如 AbsoluteLayout、FrameLayout 等布局类型,又例如 DatePicker、CalendarView 这样的功能稍复杂的通过多种控件复合而成的视图控件;而 View 就是那些功能相对单一,组成也相对单一的视图控件,例如 AnalogClock、Button、ImageView、TextView 等。

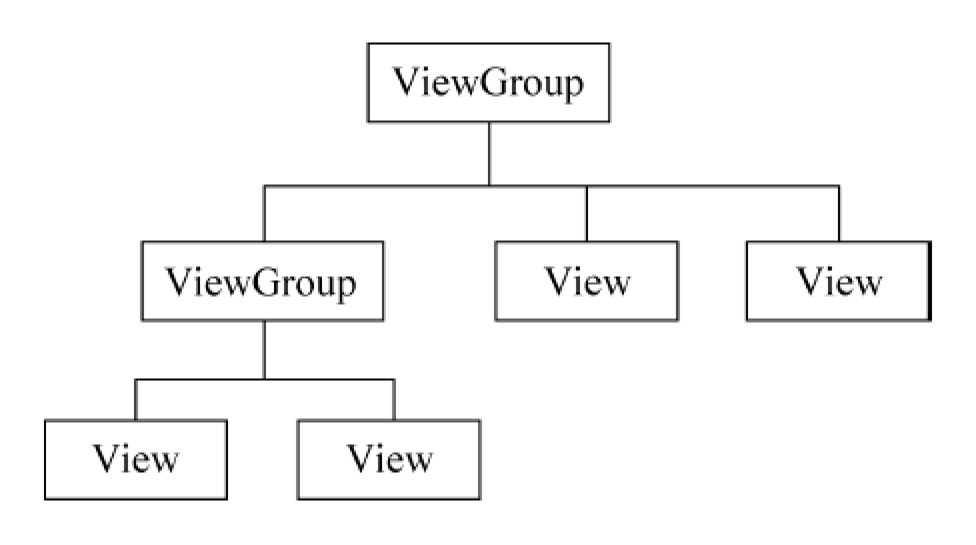


图 4-1 Android 用户界面结构

要为应用程序定制用户界面,通常有如下两种方式:

- 使用 xml 文件定制用户界面。Android 提供了使用 xml 的方式来构建视图结构,这些 xml 文件存放在项目树下的/res/layout 目录下,Android 为每种 View 都提供了很多属性,通过设置这些属性来达到定制用户界面的目的。
- 在 Java 代码中实时地对用户界面进行构建。在第一种方法中提到的用于定制 View 的那些 xml 属性,基本上每一种属性都对应了一个 Java 方法,因此,可以在 Java 代码中使用这些方法来达到同样的效果。

为了把一个视图结构显示到屏幕上,可以在对应的 Activity 中使用 setContentView()方法,并向这个方法传递一个视图结构的根节点的引用,这个引用可以是/res/layout/下的 xml 文件的 id(这个 id 由 aapt 工具自动生成),也可以是在代码中声明的 View 类对象。系统在接受此引用后就开始绘制视图。

以 xml 文件为例,在 xml 文件中的每一个视图节点都对应了一个视图控件,例如,一个<TextView>元素将在 UI 中生成一个文本视图,而一个<LinearLayout>元素将创建一个 LinearLayout 类型的视图组。

Android 提供了几种常用的布局类型,包括 AbsoluteLayout、FrameLayout、GridLayout、LinearLayout、RelativeLayout、TableLayout 等,虽然这几种基本的布局形式十分简单,但是就像乐高(LEGO)积木一样,可以通过对这几种布局类型的组合、嵌套,得到十分丰富的视图结构。

1. AbsoluteLayout(绝对布局)

在这种布局下,需要通过明确地指定各个子视图在屏幕上的 X/Y 轴坐标。但是考虑到设备的多样性,这样的布局方式显得十分不灵活并且相对于其他类型的布局更加难以确定

一个视图控件的具体位置，由于这种缺陷，在 Android 官方文档中已经将其标识成为 deprecated(过时的)，取而代之，应该使用 FrameLayout 和 RelativeLayout。AbsoluteLayout 的一个简单效果如图 4-2 所示。

2. FrameLayout(帧布局)

FrameLayout 从屏幕中截取出一块区域用于显示单独的一项视图组件，通常 FrameLayout 仅仅包含一个子视图，因为当它包含多个子视图时，很难避免在不同设备上可能出现子视图重叠的现象。子视图以栈的形式进行绘制，最近被添加的 FrameLayout 中的视图会被绘制在最上层，FrameLayout 的尺寸及可见性由其包含的最大子视图决定。FrameLayout 的一个简单效果如图 4-3 所示，在图 4-3 中可以看到，后添加的文本视图显示在先添加的按钮之上。

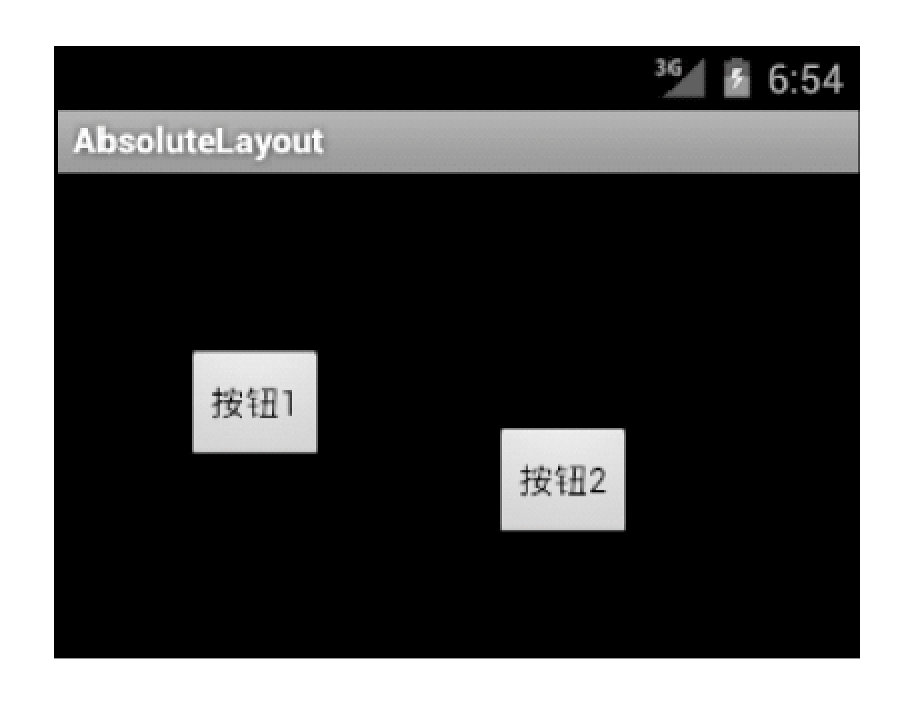


图 4-2 AbsoluteLayout 示意



图 4-3 FrameLayout 示意

3. GridLayout(网格布局)

GridLayout 是在 API Level 14 中才新加入的布局类型，它能够将一个视图按照网格的形式进行划分，并且以“格”为单位来为子视图分配空间，一个子视图可以占用一格也可以占用多格(通过属性 rowSpan 和 columnSpan 参数进行设置)，这种布局类型的加入使得多行多列的布局的实现更加高效，GridLayout 的示意图如图 4-4 所示。

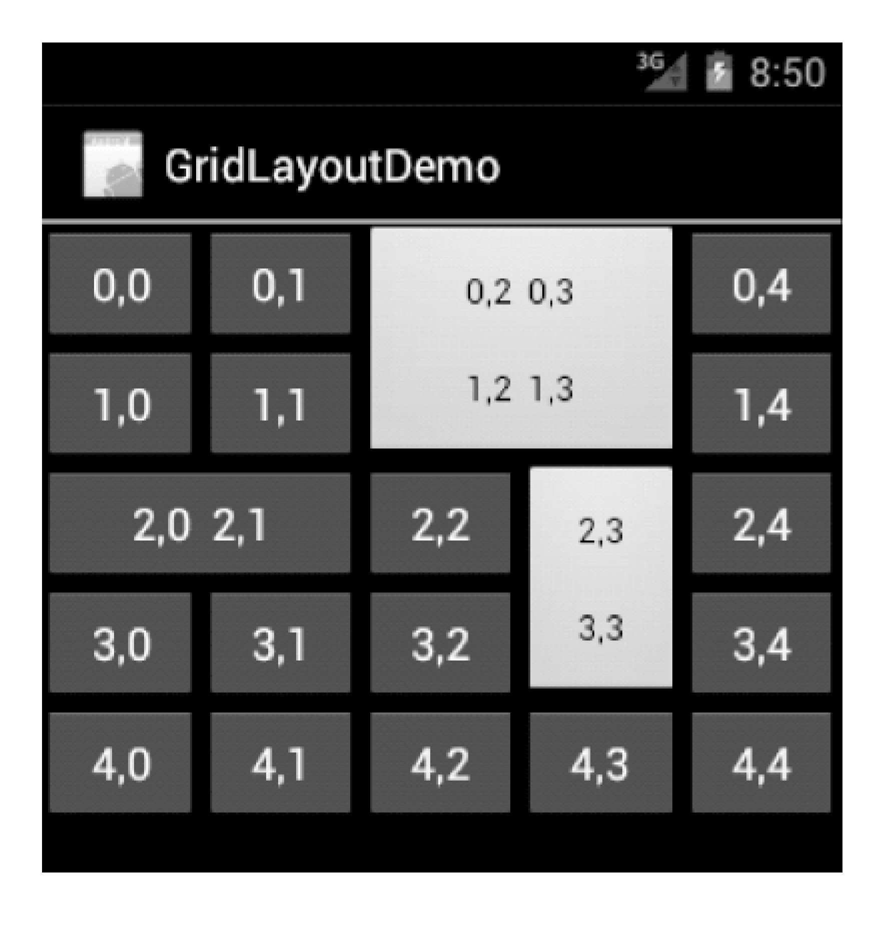


图 4-4 GridLayout 示意

4. LinearLayout(线性布局)

LinearLayout 将所有的子视图按照一列或者一行的形式进行排列，默认的排列方向是水平，可以通过 setOrientation 和 setGravity 方法或对应的属性来具体地对它的排列方式进行设置。可以通过嵌套使用 LinearLayout 来实现多行或者多列的布局，另外还可以通过 TableLayout 或 GridLayout 来实现多行和多列布局，使用何种布局方式根据具体的界面需求来确定。LinearLayout 的一个简单效果如图 4-5 所示，在这个效果中设计了 3 个 LinearLayout，其中两个 LinearLayout 嵌套在顶级 LinearLayout



内，并且它们的排列方式有所不同。

5. RelativeLayout（相对布局）

RelativeLayout 使得子视图可以使用相对关系作为参数来进行排列，常用的关系类型有：above（在某个视图上方）、below、toLeftOf、toRightOf、alignBaseLine（与某个视图的 BaseLine 对齐）、alignBottom（与某个视图底部对齐）等，示意图如图 4-6 所示。

6. TableLayout（表格布局）

该布局与 GridLayout 有些类似，它以表格的形式组织其内部的子视图，一个简单的示意图如图 4-7 所示。



图 4-5　LinearLayout 示意



图 4-6　RelativeLayout 示意



图 4-7　TableLayout 示意

4.2　控件类型

控件就是为用户界面提供服务的视图对象。Android 提供了一系列功能强大且形式丰富的控件来协助开发人员快速建立起应用程序的用户界面，从广义上来说，3.2 节所介绍的几种 Layout 也属于控件，只不过它们的功能是用于控制其内部的子视图而不具备交互的功能，就像是一个框架。Android 提供的用户界面控件分别包括了前面介绍的 Layout 和多种组件（widget），例如 Button（按钮）、TextView（文本）、EditText（文本编辑框）、ListView（列表）、CheckBox（复选框）、RadioButton（单选按钮）、Spinner（下拉列表）以及 AutoCompleteTextView（带自动补全的文本框）、图片切换器（ImageSwitcher）等。另外还有一些较复杂且常用的控件，例如时间选择器、日期选择器和缩放控件。当然，开发人员还

可以自己创建一些控件供应用程序使用，只要按照一定的标准去定义控件对象，或者直接在已有控件上进行扩展和合并。读者可以在 Android SDK 的 android. widget 包下面找到所有系统已定义好的控件。由于许多控件在使用方法上存在着共性，它们需要完成的主要任务是对用户操作的捕获与处理，因此在本节首先会介绍视图控件对用户操作的捕获和处理的方法，然后再选择性地对常用的一些控件进行介绍。

4.2.1 用户操作的捕获与处理

当各种控件被添加到应用程序用户界面中后，部分控件需要对用户的操作事件进行捕获和响应处理，例如 Button 需要响应用户点击、按下等事件，只有实现了对这些事件的处理，才能算是与用户进行了交互，在这个交互过程中就包括了响应和处理两个过程，其中响应过程就涉及 Android 对 UI 事件提供的一系列事件响应函数和回调函数，而处理过程则是这些函数中的具体实现代码。简单来说，控件捕获用户操作有如下两种方式：

- 定义一个事件监听器并将其绑定到相应的控件。用于监听用户事件，view 类包含了一系列命名类似于 On＜Action-name＞Listener 的接口，而每一个接口都提供了一个命名类似于 On＜Action-name＞()的回调方法。例如，响应视图点击事件的接口和方法分别是 View. OnClickListener 和 onClick()方法，响应触屏事件的接口和方法分别是 View. OnTouchListener 和 onTouch()方法，响应设备按键事件的接口和方法 View. OnKeyListener 和 onKey()方法等。所以如果某控件需要在它被点击时获得通知，就需要通过监听器来实现，要实现实现一个点击事件的监听器，首先这个监听器需要实现 OnClickListener 接口，并且在其 onClick 回调方法中实现具体的处理，然后通过控件的 setOnClickListener()方法将监听器绑定。
- 重写该控件相关的回调方法。这种方式允许为控件所接收到的每个事件定义默认的处理行为，并决定是否需要将事件传递给其他的子视图。

举例说明，如果要为一个按钮增加对点击事件的响应处理的功能，采用前面提到的第一种方法，可以通过如下代码实现：

```
Button button;    //定义按钮
//定义监听器
OnClickListener clickListener = new  OnClickListener(){
    public void   onClick(View v){
        //相关的处理代码
        doSomethingHere();
    }
};
//监听器绑定到按钮
button.setOnClickListener(clickListener);
```

对于第二种方法，由于控件大多数都是 android. view. View 的子类，因此可以通过重写 View 类的如下方法，来设置对这些事件的默认处理方法，例如：

- View. onKeyDown()——处理按键被按下事件。
- View. onKeyLongPress()——处理按键的长时间按下事件。
- View. onKeyUp()——处理按键弹起事件。

- View. onTouchEvent()——处理视图被触摸事件。

4.2.2 常用的一些控件

Android 提供了一些常用的控件，可以在 Eclipse 中的 ADT 插件所包含的图形化界面编辑器中直观地看到各种各样的控件，新建一个项目或者任意选择一个项目，在这个项目的/res/layout 文件夹内找到布局文件并双击打开，就能够进入这个由 Android 提供的图形化界面编辑器（如果默认没有进入这个界面，可以通过右击 xml 文件→Open With→Android Layout Editor 的方式打开），如图 4-8 所示。

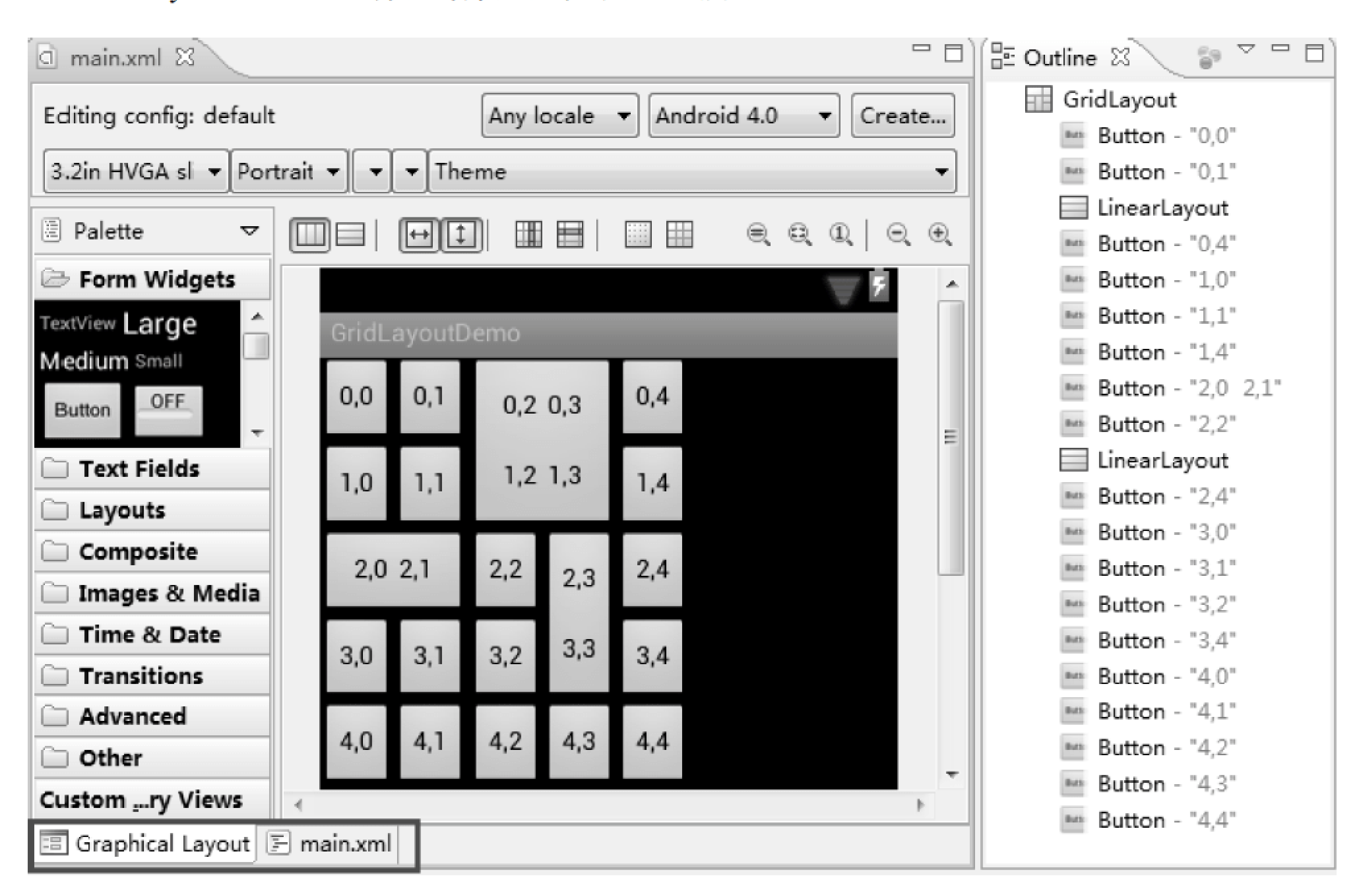


图 4-8 Android 图形化界面编辑器

如图 4-8 所示，在双击打开一个布局 xml 文件时，有两种不同的显示方式：一种就是纯文本的 xml 文件，另一种就是在图 4-8 中看到的图形化编辑界面。这两种显示方式可以通过底部方框所注明的选项卡来进行切换，通过 Android 界面编辑器能够实现 WYSIWYG（所见即所得）的设计方式。图 4-8 中右边的一栏是目前界面中所存在的一些控件的概览，可以在这个概览中右击元素来实现对某个控件的属性定义，图 4-8 中左边的一栏就是由 Android 提供的各种类型的控件，从图 4-8 中可以清晰地看到，Android 提供了如下几种类型的控件：

- Form Widgets。表单控件，包括了普通文本（TextView）、大中小的文本（Large、Medium、Small）、普通按钮（Button）、开关按钮（ToggleButton）、复选框（CheckBox）、选项按钮（RadioButton）、可被选择的文本（CheckedTextView）、Spinner（下拉列表）、不同大小的圆形进度条（ProgressBar（Large/Normal/Small））、水平进度条（ProgressBar（Horizontal））、拖动条（SeekBar）、联系人标记

(QuickContactBadge)、单选组框(RadioGroup)、评分栏(RatingBar)。

- Text Fields。文本域,这些类型实际上都是由 EditText 控件定制而来,包括了纯文本框(Plain Text)、人名文本框(Person name)、密码文本框(Password)、纯数字密码文本框(Password(Numeric))、电子邮件地址文本框(E-mail)、电话号码文本框(Phone)、邮寄地址文本框(Postal Address)、多行文本框(Multiline Text)、时间文本框(Time)、日期文本框(Date)、数字文本框(Number)、带符号的数字文本框(Number(signed))、带小数的数字文本框(Number(Decimal))、自动完成文本框(AutoCompleteTextView)、通过逗号分隔的多子字符串自动完成文本框(MultiAutoCompleteTextView)。
- Layouts。布局结构,即 4.1 节提到的内容。还包括了 Include Other Layout(包含其他的 xml 布局文件)、Fragment(能够将部分用户界面作为一个片段放置到其他 Activity 中的布局结构)、Space(为用户界面各控件之间增加间隔)。
- Composite。符合控件,包括了 ListView(列表视图)、ExpandableListView(可分级展开的列表视图)、GridView(网格视图)、ScrollView(滚动视图)、HorizontalScrollView(水平滚动视图)、SearchView(搜索视图)、SlidingDrawer(滑动式抽屉)、TabHost(标签窗口视图容器)、TabWidget(标签窗口视图的标签)、WebView(网页视图)。
- Image&Media。图像和媒体控件,包括了 ImageView(图像视图)、ImageButton(图像按钮)、Gallery(相册)、MediaController(媒体播放控制器)、VideoView(视频视图)。
- Time&Date。时间和日期控件,包括了 TimePicker(时间选择器)、DatePicker(日期选择器)、CalendarView(日历视图)、Chronometer(计时器)、AnalogClock(模拟时钟)、DigitalClock(数字时钟)。
- Transitions。切换器,包括了 ImageSwitcher(图像切换器)、AdapterViewFlipper(带翻转动画效果的切换器,通过 Adapter 为该切换器关联资源)、StackView(层叠切换器)、TextSwitcher(文本切换器)、ViewAnimator(为视图切换提供动画效果的切换器)、ViewFlipper(类似 AdapterViewFlipper,常用于 FrameLayout 内)、ViewSwitcher(视图切换器)。
- Advanced。较高级的控件,包括了 requestFocus(为其父视图或者后代视图请求焦点)、View(视图)、ViewStub(视图代理,使得应用程序在运行时可以动态地加载 xml 布局文件)、GestureOverlayView(手势识别覆盖层,一个透明的控件,可位于其他的控件之上或者包含其他的控件)、TextureView(纹理视图,用于显示一个内容流,例如视频流或 OpenGL 内容流,只能够在硬件加速窗口中使用,否则不会绘制任何图形)、SurfaceView(一块专用于绘图的区域,在非 UI 线程中进行绘制)、NumberPicker(数字选择器)、ZoomButton(缩放按钮)、ZoomControls(缩放控制器)、DialerFilter(拨号过滤器)、TwoLineListItem(两行的列表单元)。

4.3 通知消息

当应用程序的状态发生变化或者出现其他一些事件的时候，就需要通知用户，并且把相应的信息呈现给用户，这些信息便于用户了解应用程序发生了什么事件。如果需要用户进行选择，那么这些信息还必须要阐明这些选择的目标和条件。为此，需要借助 Android 的事件通知机制来实现这些目标。Android 为通知消息提供了 3 种形式，每种形式都有自己所适用的场合和优势，本节将分别对这 3 种通知消息进行介绍，并通过示例来具体说明每一种通知消息的使用方法。

4.3.1 浮出消息(Toast)

浮出消息，类似于 Windows 操作系统中的弹出气球通知，Toast 通知是一种从屏幕背景逐渐浮现出通知而后又渐渐消失的一种形式，如图 4-9 所示是系统默认的 Toast 消息形式，也可以自己设置 Toast 消息的表现形式，如图 4-10 所示。Toast 消息不会影响现有的界面，也不可被交互，它还可以被一个 Service 呼出，因为它不依赖于其他界面。Toast 消息常用于一些简单的提示和状态汇报，例如文件保存完毕等事件，如果需要与用户交互，应该使用后面提到的 Status Bar Notification。



图 4-9 默认 Toast 通知消息形式



图 4-10 自定义 Toast 通知消息形式

1. 使用默认 Toast 通知消息

要使用系统默认形式的 Toast 通知消息，通过简单的一行代码即可实现，本例中直接在新建的 HelloWorld 项目中添加了一个 Button，使得能够通过单击按钮来显示一条 Toast 通知消息，代码如下：

```
01 public class HelloWorld extends Activity {
02     @Override
03     public void onCreate(Bundle savedInstanceState) {
04         super.onCreate(savedInstanceState);
05         setContentView(R.layout.main);
06         Button buttonToast = (Button) findViewById(R.id.button1);
07         buttonToast.setOnClickListener(new OnClickListener() {
08          @Override
09          public void onClick(View v) {
10              Toast.makeText(HelloWorld.this, "Toast 消息", Toast.LENGTH_LONG).show();
11          }
12        });
13     }
14 }
```

在上面的代码中，第 10 行的代码即实现了 Toast 消息的显示。其中 Toast 类的静态方法 makeText()用于构造一条默认风格的 Toast 消息，方法原型如下：

```
static Toast makeText(Context context, CharSequence text, int duration)
```

参数列表：

- context——应用程序上下文。
- text——需要显示的内容。
- duration——消息显示的时间长短，只能为 LENGTH_SHORT 或 LENGTH_LONG 这两个值之一，两者分别代表了较短和较长的时间。

返回值：

- Toast——一条构造好的 Toast 消息对象。

2. 自定义风格的 Toast 通知消息

要自定义 Toast 通知消息的风格，可以使用 Toast.setView()方法为 Toast 消息设置一个视图，因此，主要的工作是为 Toast 消息定制一个消息视图，为此，在 res/layout/下添加一个用于表示 Toast 消息的布局形式的 toast_layout.xml 文件，内容如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toast_layout_root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#DAAA"
    android:orientation="horizontal"
    android:padding="10dp" >
    <ImageView
        android:id="@+id/image"
        android:src="@drawable/toast_image"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="10dp" />
    <TextView
        android:id="@+id/text"
```

```
16          android:layout_width = "wrap_content"
17          android:layout_height = "fill_parent"
18          android:gravity = "center_vertical"
19          android:text = "自定义 Toast 通知消息"
20          android:textColor = "#FFF" />
21  </LinearLayout>
```

该布局文件采用了 LinearLayout 作为根布局，子视图的排列方向是水平方向（第 06 行），并且设置了背景颜色（第 05 行），还有子视图与父视图边界的距离（第 07 行），该布局具有两个水平排列的视图：一个 ImageView（第 08～13 行）和一个 TextView（第 14～20 行）。定义好了自定义的 Toast 布局风格之后，在 Activity 中只需要将之前的：

```
Toast.makeText(HelloWorld.this, "Toast 通知消息", Toast.LENGTH_LONG).show();
```

这一行代码，替换为如下代码即可：

```
01          LayoutInflater inflater = getLayoutInflater();
02          View layout = inflater.inflate(R.layout.toast_layout,
03              (ViewGroup) findViewById(R.id.toast_layout_root));
04          Toast toast = new Toast(getApplicationContext());
05          toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
06          toast.setDuration(Toast.LENGTH_LONG);
07          toast.setView(layout);
08          toast.show();
```

上述代码设置了用于填充 Toast 消息的视图（第 01～03、第 07 行），并且设置了消息在屏幕中的显示位置（第 05 行）以及显示的时间长短（第 06 行）。这样就实现了如图 4-10 所示的效果。

4.3.2 顶部状态通知栏（Status Bar Notification）

Status Bar Notification 将会在系统的状态栏上显示一条消息，这种消息的默认样式会包含一个具有鲜明含义的图标并且伴随着一段文字（见图 4-11），通常还有一条扩展信息（通过拉下任务栏可以看到），如图 4-12 所示，这条扩展信息是能够与用户进行交互的，通过点击它可以触发特定的事件（如启动一个 Activity，如图 4-13 所示）。另外，可以在 Status Bar 消息弹出的同时伴随声音、振动或者灯光提示。Status Bar 消息通常在应用程序以服务的形式在后台运行并且遇到一些需要用户处理的事件时呼出，需要注意的是，后台服务永远不应该被设计成为能够自动地去启动一个 Activity，这会造成十分糟糕的用户体验，就像在 Windows 系统中的弹出窗口（Activity 就好比一个全屏的弹出窗口）一样令用户厌恶。另外，如果当前应用程序在前台运行，更好的方式是使用后面介绍的 Dialog Notification。

1. 基本实现方法

实现一个普通风格的 status bar 通知消息通常需要经过如下 4 步：

（1）获取 NotificationManager，该对象用于管理系统的 status bar 通知消息，可以通过如下方法获取：



图 4-11 StatusBar 新消息

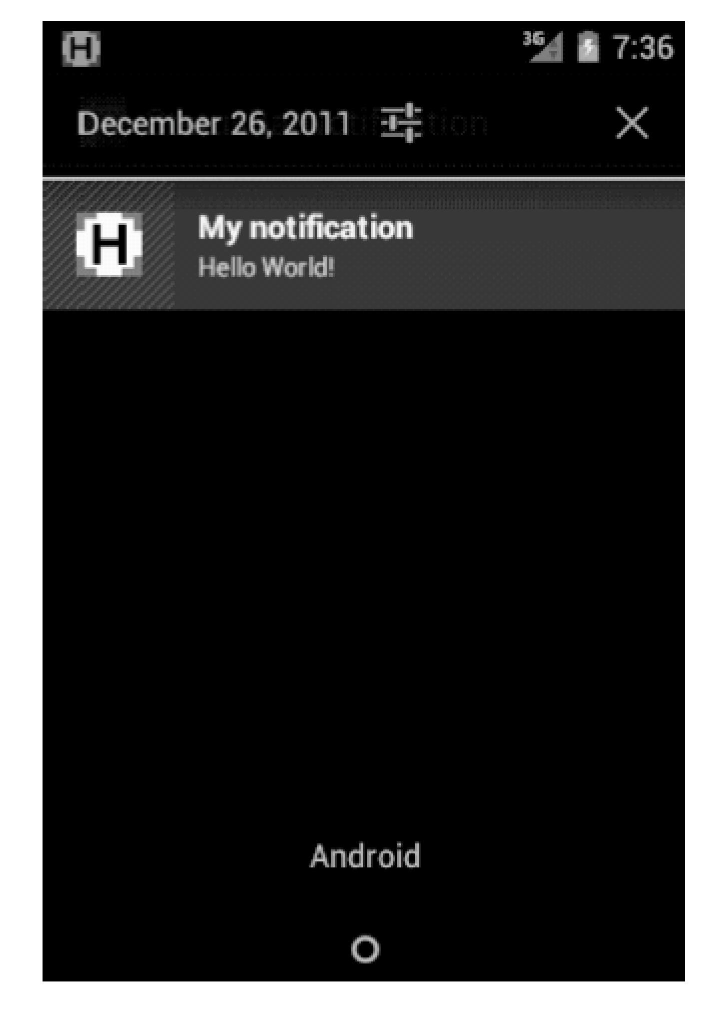


图 4-12 下拉扩展信息



图 4-13 点击扩展信息弹出新 Activity

```
private NotificationManager mNotificationManager;
mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
```

实例化一个 Notification 对象，这个对象代表一条具体的 statusbar 通知消息：

```
CharSequence tickerText = "Hello";
long when = System.current TimeMillis();
Notification notification = new Notification(R.drawable.statusbar_icon, tickerText,
when);
```

（2）如果需要，可以为这个 Notification 定义详细消息，还可以为其定义点击后产生的 Intent：

```
Context context = getApplicationContext();
CharSequence contentTitle = "My notification";
CharSequence contentText = "Hello World!";
Intent notificationIntent = new Intent(this, StatusBarNotificationActivity.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
notification.setLatestEventInfo(context, contentTitle, contentText, contentIntent);
```

（3）最后，使用 NotificationManager 对这个 Notification 对象进行发布，其中需要为该消息指定一个唯一的 ID：

```
private static final int HELLO_ID = 1;
mNotificationManager.notify(HELLO_ID, notification);
```

2. 更新 Notification

通常需要对 Notification 进行实时更新，例如当一条新的短信到来时，系统会在

statusbar中发布一条消息，当第二条新短信到来时，如果第一条短信仍然没有被用户读取，那么通常的做法是在 Notification 中显示最新的一条短信摘要，并且更新未读消息的计数，这种实现方式可以避免 Notification 过多而造成混乱。

更新消息显示通过 setLatestEventInfo()方法来实现，由于每一条 Notification 都被指定了一个唯一的 ID，因此只需要调用 notify()方法即可刷新消息显示。

3. 丰富 Notification 的表现形式

为了使得 Notification 更有表现力，可以为其增加声音、振动、闪光等特性，例如，增加声音提示功能可以通过为 Notification 的 defaults 域赋值来实现，defaults 域的值就决定了该条 Notification 的一些特性：

```
notification.defaults |= Notification.DEFAULT_SOUND;
```

上面的代码为 notification 增加了默认的提示音，如果想使用自定义的提示音，也可以通过如下两种方式来为 defaults 域赋值：

```
notification.sound = Uri.parse("file:///sdcard/notification/ringer.mp3");
notification.sound = Uri.withAppendedPath(Audio.Media.INTERNAL_CONTENT_URI, "6");
```

其中第一种方式通过 Uri 的方式指定了外部的音频文件，而第二种方式则使用了系统的音频文件 ContentProvider。

如果要为 Notification 增加振动提示功能，可以通过如下两种方式：

```
notification.defaults |= Notification.DEFAULT_VIBRATE;
long[] vibrate = {0,100,200,300};
notification.vibrate = vibrate;
```

其中第一种方式(第 1 行)使用了默认的振动方式，第二种方式(后两行)则是使用了自定义的振动方式。

相似地，要增加闪光功能，可以通过如下两种方式：

```
notification.defaults |= Notification.DEFAULT_LIGHTS;
notification.ledARGB = 0xff00ff00;
notification.ledOnMS = 300;
notification.ledOffMS = 1000;
notification.flags |= Notification.FLAG_SHOW_LIGHTS;
```

另外，Notification 还支持如下一些特性：

- 当用户点击 Notification 后自动取消显示，通过 FLAG_AUTO_CANCEL 标志确定。
- 循环播放音效直到用户进行操作，通过 FLAG_INSISTENT 标志确定。
- 用于表示该消息所代表的事件仍然在继续执行，通过 FLAG_ONGOING_EVENT 标志确定。
- 防止消息被 Clear notification 按钮所清除，通过标志 FLAG_NO_CLEAR 标志确定。

4. 自定义 Notification 风格

与 Toast 一样，Status Bar Notification 同样可以自定义其显示样式，不同的是，这里需要使用 RemoteView 来实现，通过如下方式来自定义样式：

```
RemoteViews contentView = new RemoteViews(getPackageName(),
R.layout.custom_notification_layout);
contentView.setImageViewResource(R.id.image, R.drawable.notification_image);
contentView.setTextViewText(R.id.title, "Custom notification");
contentView.setTextViewText(R.id.text, "This is a custom layout");
notification.contentView = contentView;
```

其中 R.layout.custom_notification_layout 就是用于自定义消息样式的布局文件。

4.3.3 对话框（Dialog）

Android 还提供了一种非常好的与用户交互的对象——对话框（Dialog），对话框通常是覆盖在当前 Activity 上面的小窗口，当对话框出现后，当前的 Activity 将失去焦点，用户在此情况下只能与弹出的对话框进行交互。Android 对话框能够十分方便地进行创建、保存、回复和管理。使用对话框时会接触到很多的方法，如 onCreateDialog（int ID）、onPrepareDialog(int ID, Dialog dialog)、showDialog(int ID)和 dismissDialog(int ID)等，下面介绍一下这些方法的作用及用法。

- onCreateDialog(int ID)通过传入的 ID 号用以生成一个指定的 Dialog 对象，当 showDialog(int ID)方法执行时会触发这个方法。
- onPrepareDialog(int ID, Dialog dialog)是一个可选方法，它可以在 Dialog 对象已经生成，但是还没有显示之前对这个 Dialog 对象进行所需要的修改，如修改标题、显示内容等。
- showDialog(int ID)方法用于显示 ID 所对应的 Dialog 对象，如果这个方法被调用，则会触发回调方法 onCreateDialog(int ID)。
- dismissDialog(int ID)方法用于关闭 ID 对应的 Dialog 对象在 Activity 中的显示（如果要完全销毁 Activity 中的某个对话框对象，让它再也不显示，可以调用 removeDialog(int ID)方法）。

其中，onCreateDialog（int ID）和 onPrepareDialog（int ID，Dialog dialog）这两个方法是 Dialog 比较常用到的回调方法。在调用了 showDialog（int ID）之后，如果对应 ID 号的 Dialog 对象是第一次生成，系统就回调 onCreateDialog（int ID）方法，然后再调用 onPrepareDialog(int ID,Dialog dialog)方法，若对应 ID 号的 Dialog 对象已经生成，只是没有被显示，则系统直接回调 onPrepareDialog(int ID,Dialog dialog)。

Android 提供了 AlertDialog、TimePickerDialog、DatePickerDialog、ProgressDialog 等多种形式的 Dialog，其中最常用的是 AlertDialog 和 ProgressDialog。

AlertDialog 允许在对话窗口中添加最多 3 个按钮（positive、neutral、negative），还可以包含一个带选项的列表（如 CheckBoxes 或者 RadioButtons 等），它是实现 Android 中大多数对话框的 Dialog 直接子类之一，使用 AlertDialog 对象通常不会通过构造函数来创建，而

是通过其内部静态类 AlertDialog.Builder 来进行构造。简单地说,新建一个带有确认按钮的对话框对象的语法为:

```
new AlertDialog.Builder(this).setMessage("string")
    .setPositiveButton("name",new DialogInterface
    .OnClickListener(){public void onClick(DialogInterface dialog, int whichButton){}}).
create();
```

如上面的代码所示,可通过一连串方法链的方式为新建对象设置相关属性,通过内部类的方式设置按钮点击事件的回调方法 onClick()。

要创建一个包含列表的对话框,可以通过如下形式来实现:

```
new AlertDialog.Builder(this)
    .setItems(CharSequence[] items,
        new DialogInterface.OnClickListener(){
            public void onClick(DialogInterface dialog, int  whichButton){}})
    .create();
```

其中,setItem()方法的第一个参数代表了列表的内容,如果需要创建单选对话框或者多选对话框,可以使用 setSingleChoiceItems()方法或者用 setMultiChoiceItems()方法代替 setItems()方法即可。

ProgressDialog 是 AlertDialog 的扩展类,它在 AlertDialog 的基础上加入了表示进度的功能,通常表现为包含有进度条的对话框,因为其扩展于 AlertDialog,ProgressDialog 的基本功能和用法与 AlertDialog 基本一样。默认创建圆圈形状的进度条,使用如下语句即可:

```
new ProgressDialog.setTitle(String).setMessage(String)
```

如果需要创建条形的进度条,可以使用 setProgressStyle()方法来设置进度条的风格为条形,创建该类对话框的代码如下:

```
new ProgressDialog(context).setProgressStyle().setCancelable()
```

对话框在创建后,可以使用 dialog.setProgress(int)方法来设置进度条的进度,或者使用 dialog.incrementProgressBy(int)方法在当前进度的基础上进行增加。

TimePickerDialog 和 DatePickerDialog 这两种对话框的使用方法比较简单,直接使用相应的构造方法构造即可,类似于普通控件的用法,因此这里就不再赘述。

接下来通过示例来介绍 4 种对话框的实现。首先,在 Activity 中声明 4 个常量分别用于指代 4 种对话框:

```
private static final int DIALOG_WITH_3_BUTTONS = 1;
private static final int DIALOG_SINGLE_CHOICE = 2;
private static final int DIALOG_WITH_EDITTEXT = 3;
private static final int DIALOG_WITH_PROGRESS = 4;
```

然后,在 onCreate()方法中将 Activity 的界面视图中的 4 个按钮分别绑定点击事件监听器,

并且分别调用 showDialog(int)方法：

```
Button button1 = (Button)findViewById(R.id.button1);
button1.setOnClickListener(new OnClickListener(){
    public void onClick(View v) {
        showDialog(DIALOG_WITH_3_BUTTONS);
    }
});
Button button2 = (Button)findViewById(R.id.button2);
button2.setOnClickListener(new OnClickListener(){
    public void onClick(View v) {
        showDialog(DIALOG_SINGLE_CHOICE);
    }
});
Button button3 = (Button)findViewById(R.id.button3);
button3.setOnClickListener(new OnClickListener(){
    public void onClick(View v) {
        showDialog(DIALOG_WITH_EDITTEXT);
    }
});
Button button4 = (Button)findViewById(R.id.button4);
button4.setOnClickListener(new OnClickListener(){
    public void onClick(View v) {
        showDialog(DIALOG_WITH_PROGRESS);
    }
});
```

再实现用于响应 showDialog()方法的 onCreateDialog()方法，用于生成对应于前面声明的常量的对话框实例：

```
//生成对话框
public Dialog onCreateDialog(int id){
    switch(id){
    case DIALOG_WITH_3_BUTTONS:
        return dialogWith3Buttons(DialogNotificationActivity.this);
    case DIALOG_SINGLE_CHOICE:
        return dialogSingleChoice(DialogNotificationActivity.this);
    case DIALOG_WITH_EDITTEXT:
        return dialogWithEditText(DialogNotificationActivity.this);
    case DIALOG_WITH_PROGRESS:
        return dialogWithProgress(DialogNotificationActivity.this);
    }
    return null;
}
```

在 onCreateDialog()方法中使用了一系列的用于创建对话框的方法，这些方法分别完成一种类型对话框的创建。

首先来看一下 dialogWith3Buttons()方法的实现，这个方法用于生成一个具有确定、提示和取消这 3 个按钮的对话框，代码如下：

```
private Dialog dialogWith3Buttons(Context context){
 //带 3 个按钮的对话框
 AlertDialog.Builder dialogWith3Buttons = new AlertDialog.Builder(context);
 dialogWith3Buttons.setIcon(R.drawable.icon);
 dialogWith3Buttons.setTitle("带有 3 个按钮的对话框");
 dialogWith3Buttons.setMessage("3 个按钮的对话框演示");
 dialogWith3Buttons.setPositiveButton("确定", new DialogInterface.OnClickListener(){
        public void onClick(DialogInterface dialog, int which) {
            setTitle("点击了对话框的确定按钮");
        }
 });
 dialogWith3Buttons.setNeutralButton("提示", new DialogInterface.OnClickListener(){
        public void onClick(DialogInterface dialog, int which) {
            setTitle("点击了对话框的提示按钮");
        }
 });
 dialogWith3Buttons.setNegativeButton("取消", new DialogInterface.OnClickListener(){
        public void onClick(DialogInterface dialog, int which) {
            setTitle("点击了对话框的取消按钮");
        }
 });
    return dialogWith3Buttons.create();}
```

从上面的代码中可以看到，这 3 个按钮还分别设置了处理点击事件的 onClick()方法，作用是在点击了按钮之后更新 Activity 的标题显示，这个对话框的表现形式如图 4-14 所示。

实现带 3 个选项的单选对话框的代码如下：

```
private Dialog dialogSingleChoice(Context context){
 //单选项对话框
 final String[] str = {"赞同","反对","弃权"};
 AlertDialog.Builder dialogSingleChoice = new AlertDialog.Builder(this)
        .setTitle("单选对话框").setIcon(R.drawable.icon)
        .setSingleChoiceItems(str, 0, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                dialog.dismiss();
                setTitle("您选择的是" + str[which]);
            }
        }).setNegativeButton("取消", null);
 return dialogSingleChoice.create();}
```

对应的效果如图 4-15 所示。

实现带编辑框(通过布局文件，类似于自定义 Toast)的对话框的代码如下：

```
private Dialog dialogWithEditText(Context context){
    //通过.xml 自定义对话框布局
    LayoutInflater li = LayoutInflater.from(this);
    final View edit = li.inflate(R.layout.edit_dialog, null);
```

```
        AlertDialog.Builder dialogWithEditText = new AlertDialog.Builder(context);
        dialogWithEditText.setIcon(R.drawable.icon).setTitle("可输入对话框").setView(edit);
        dialogWithEditText.setPositiveButton("确定", null).setNegativeButton("取消", null);
        return dialogWithEditText.create();
    }
```

其中使用到了 edit_dialog.xml,这个文件的内容用于生成一个带文本输入框的布局。该对话框的效果如图 4-16 所示。



图 4-14 带 3 个按钮的对话框



图 4-15 单选对话框



图 4-16 带文本输入的对话框

最后一种对话框就是带进度条的对话框,代码如下:

```
    private Dialog dialogWithProgress(Context context){
     //进度条对话框
     final ProgressDialog dialogWithProgress = new ProgressDialog(context);
     dialogWithProgress.setTitle("进度条对话框");
     dialogWithProgress.setMessage("正在链接");
     dialogWithProgress.setButton("取消", new DialogInterface.OnClickListener(){
             public void onClick(DialogInterface dialog, int which) {
                 dialogWithProgress.dismiss();//点击取消按钮时,对话框消失
             }
     });
        return dialogWithProgress;
    }
```

默认的进度条是圆圈形状的,如图 4-17 所示。

如果想使用带进度显示的进度条,只要添加如下代码即可:

```
dialogWithProgress.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
```

效果如图 4-18 所示。



图 4-17　进度条对话框



图 4-18　条形进度条对话框

4.4　菜单(Menu)

本节将介绍用户界面中另一个十分重要的组成部分——菜单。在 Android 中,菜单分为 3 种:选项菜单(OptionsMenu)、上下文菜单(ContextMenu)和子菜单(SubMenu)。下面分别对这 3 种菜单进行介绍。

4.4.1　选项菜单

Android 模拟器提供了虚拟键盘,键盘中包含了各种常用的操作键,选项菜单就是使用 Menu 键来弹出的,当点击 Menu 键时,每个 Activity 都可以对这一事件作出相应的处理,如果有响应则会在当前屏幕的下面弹出一个菜单,这个就是选项菜单(OptionsMenu),显示当前 Activity 的可用操作列表。

在选项菜单的创建中,最主要的两个方法是 onCreateOptionsMenu(Menu menu)和 onOptionsItemSelected(MenuItem item),其中前一个方法表明如何创建菜单,后一个方法则是用于设置各菜单选项被选择后将会进行的动作。选项菜单还有另外两个方法分别是 onOptionsMenuClose(Menu menu)和 onPrepareOptionsMenu(Menu menu),前一个方法在菜单被关闭时触发(能触发菜单关闭的动作有 3 个:再次单击 Menu 键、单击 Menu 旁边的 Back 键,或者选择菜单中的某个选项),后一个方法在选项菜单弹出前被触发。

实现按 Menu 键呼出菜单的 onCreateOptionsMenu()方法代码如下:

```
public boolean onCreateOptionsMenu(Menu menu){
    menu.add(Menu.NONE, Menu.FIRST + 1,1,"添加").setIcon(android.R.drawable.ic_menu_add);
    menu.add(Menu.NONE, Menu.FIRST + 2,2,"删除")
        .setIcon(android.R.drawable.ic_menu_delete);
    menu.add(Menu.NONE, Menu.FIRST + 3,3,"保存")
        .setIcon(android.R.drawable.ic_menu_edit);
```

```
menu.add(Menu.NONE,Menu.FIRST + 4,4,"帮助")
        .setIcon(android.R.drawable.ic_menu_help);
    menu.add(Menu.NONE,Menu.FIRST + 5,5,"详情")
            .setIcon(android.R.drawable.ic_menu_info_details);
return true;
}
```

如上面的代码所示，通过 menu 的 add()方法为菜单添加了若干个菜单项，menu.add()方法有 4 个参数，分别为：groupId(组别)，表示该选项所属哪个组，如果没有分组，可设置为 menu.NONE 或者 0；itemId(选项 ID)，这个参数代表菜单项的 id，系统根据这个 id 来确定被操作的菜单项；order(顺序)，表示某一项在整个菜单中的位置；title(文本)，表示某一项要显示的文本。然后使用 setIcon()方法为每个菜单项设置了图标，这里通过 android.R.drawable.…的方式直接使用系统图标，也可以通过 R.drawble.…的方式使用当前项目下的图标资源。需要注意的是该方法的返回值，仅当返回值为 true 时，这个菜单才会显示，否则点击 Menu 键不会有菜单弹出。

每一个菜单项被选择后所发生的事件处理在 onOptionsItemSelected()方法中实现：

```
public boolean onOptionsItemSelected(MenuItem item){
    switch(item.getItemId()){
    case Menu.FIRST + 1:
        doSomething();
        break;
    case Menu.FIRST + 2:
        doSomething();
        break;
    case Menu.FIRST + 3:
        doSomething();
        break;
    case Menu.FIRST + 4:
        doSomething();
        break;
    case Menu.FIRST + 5:
        doSomething();
        break;
    }
    return true;
}
```

如上面的代码所示，通过 item.getItemId()方法来确定是哪一个菜单项被选择，从而做出相应的处理，返回值为 true 的含义是该事件已经处理完成。菜单弹出的效果如图 4-19 所示。

4.4.2 上下文菜单

相信读者对 PC 上右击的操作非常熟悉，Android 中的上下文菜单就类似于 PC 的右键弹出菜单，当为某个控件注册上下文菜单后，长时间按下该控件就会弹出一个菜单。Android 中任何控件都可注册上下文菜单，比较常见的使用场合是 ListView 中的列表项。

使用上下文菜单需要使用到 3 个方法。包括了建立并添加上下文菜单项的

onCreateContextMenu()方法、响应上下文菜单项单击的onContextItemSelected()方法和为某个控件注册上下文菜单的registerForContextMenu()方法。前两个方法与前面的OptionMenu中对应方法的使用类似。

长时间按下某控件呼出上下文菜单的onCreateContextMenu()方法代码如下。

```
public void onCreateContextMenu(ContextMenu menu, View
view,ContextMenuInfo menuInfo){
    menu.setHeaderTitle("上下文菜单");
    menu.add(0, menu.FIRST + 1, 0, "菜单项 1");
    menu.add(0, menu.FIRST + 2, 0, "菜单项 2");
    menu.add(0, menu.FIRST + 3, 0, "菜单项 3");
}
```



图 4-19 OptionsMenu 菜单

上下文菜单中每一项被点击的事件响应代码在onContextItemSelected()方法中：

```
public boolean onContextItemSelected(MenuItem item){
    switch(item.getItemId()){
    case Menu.FIRST + 1:
        Toast.makeText(this, "选择了菜单项 1", Toast.LENGTH_LONG).show();
        break;
    case Menu.FIRST + 2:
        Toast.makeText(this, "选择了菜单项 2", Toast.LENGTH_LONG).show();
        break;
    case Menu.FIRST + 3:
        Toast.makeText(this, "选择了菜单项 3", Toast.LENGTH_LONG).show();
        break;
    }
    return true;
}
```



图 4-20 上下文菜单

这里用于演示，在选择某一项之后就会弹出一个Toast消息。最后还需要为控件注册这个上下文菜单，即使用registerForContextMenu()方法，通常用在onCreate()方法中：

```
TextView tv;
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.contextmenu);
    tv = (TextView)findViewById(R.id.textView1);
    registerForContextMenu(tv);//为文本框注册上下文
    //菜单
}
```

上下文菜单的效果如图 4-20 所示。

4.4.3 多级菜单

多级菜单实际上是一种特殊的上下文菜单，它的一级菜单本身就是一个上下文菜单，不同的是它的每一个一级菜单选项又都是一个上下文菜单，这样就构成了层层递进的菜单。使用子菜单的方法与上下文菜单的几个方法是一样的。这里通过一个具有二级子菜单的示例代码来进行说明，该二级菜单的 onCreateContextMenu()方法代码如下：

```
public void onCreateContextMenu(ContextMenu menu,View view,ContextMenuInfo menuInfo){
    menu.setHeaderTitle("SubMenu - 一级菜单");
    SubMenu sub1 = menu.addSubMenu("菜单 1");
    sub1.add(0,sub1.FIRST + 1,0,"子菜单项 1");
    sub1.add(0, sub1.FIRST + 2, 1, "子菜单项 2");
    SubMenu sub2 = menu.addSubMenu("菜单 2");
    sub2.add(0,sub2.FIRST + 3,0,"子菜单项 3");
    sub2.add(0, sub2.FIRST + 4, 1, "子菜单项 4");
}
```

代码实现了两级菜单，通过 SubMenu 实现了二级菜单。一级菜单项被点击时会自动跳转到对应的二级菜单，因此在 onContextItemSelected()方法中只需要处理二级菜单项：

```
public boolean onContextItemSelected(MenuItem item){
    switch(item.getItemId()){
    case SubMenu.FIRST + 1:
        //设置子菜单中的第一个子菜单的第一个选项的响应事件
        Toast.makeText(this, "子菜单项 1", Toast.LENGTH_LONG).show();
        break;
    case SubMenu.FIRST + 2:
        Toast.makeText(this, "子菜单项 2", Toast.LENGTH_LONG).show();
        break;
    case SubMenu.FIRST + 3:
        Toast.makeText(this, "子菜单项 3", Toast.LENGTH_LONG).show();
        break;
    case SubMenu.FIRST + 4:
        Toast.makeText(this, "子菜单项 4", Toast.LENGTH_LONG).show();
        break;
    }
    return true;
}
```

最后，同样需要为控件注册该菜单：

```
public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.submenu);
     TextView tv2 = (TextView)findViewById(R.id.textView2);
     registerForContextMenu(tv2);//为文本框注册子菜单
}
```

该菜单的效果如图 4-21 和图 4-22 所示。



图 4-21　一级菜单

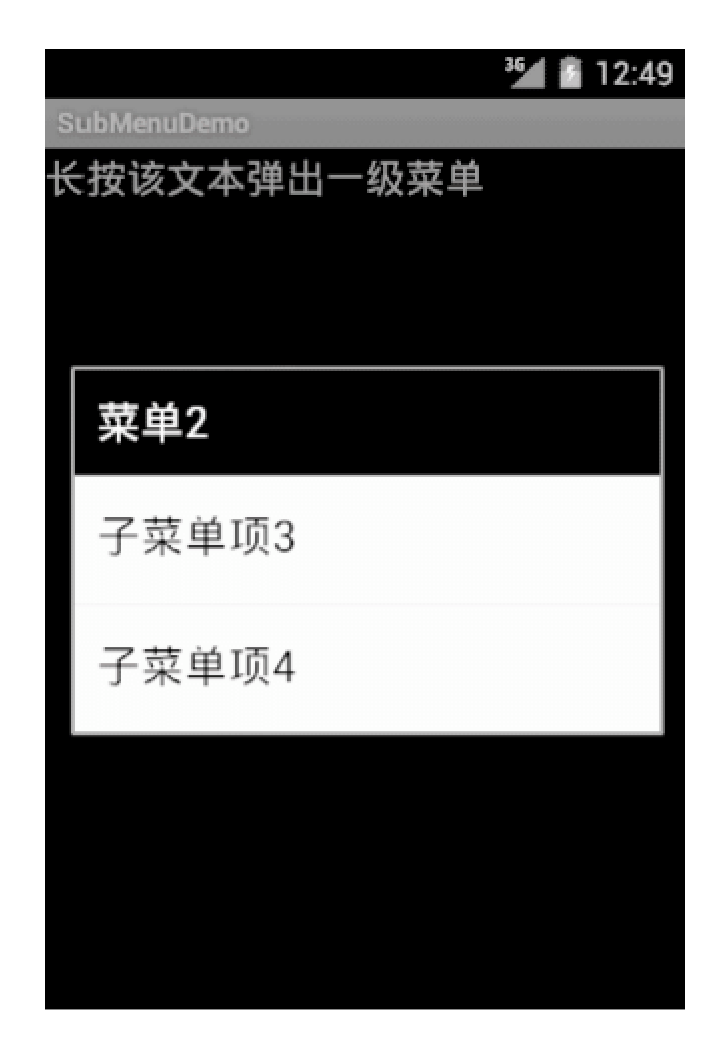


图 4-22　二级菜单

4.5　App Widget(桌面小插件)

4.5.1　App Widget 简介

Widget 是 Android 1.5 以后加入的一个特性,允许程序显示一些常用而又重要的信息在用户的 Home screen(桌面主屏)上。

简单地说,App Widget 具有两个特点:一是可以添加到 Home screen 上,二是可以按照一定的频率对内容进行更新。App Widgets 由 3 个主要的部分组成:

- 边界,用于指定其在 Home Screen 上的位置、大小。
- 边框,可以理解为图形化的边界。
- Widget 的图形控件和其他元素。

与 Status Bar Notification 相同,App Widget 的界面显示依赖于 RemoteView 类,因此仅支持一部分布局形式和部分控件,包括了如下一些:

- 布局形式——FrameLayout、LinearLayout、RelativeLayout。
- 控件类型——AnalogClock、Button、Chronometer、ImageButton、ImageView、ProgressBar、TextView。

App Widgets 是一种最小化的应用程序的可视窗口,这些应用程序可以嵌入在其他的应用程序(Home Screen)中并接收定期更新,一个能够拥有其他的 App Widgets 的应用程序组件被称为一个 App Widgets host(典型的就是 Home Screen)。开发人员通过 AppWidgetProvider 向系统提供自己开发的 App Widgets。

4.5.2　App Widget 示例

要开发自己的 App Widget,需要了解如下几点:

- AppWidgetproviderInfo 类(用于描述 App Widget 的基本信息)——它是 APP Widget 在 XML 中定义的一个元数据,比如应用程序的窗口布局、更新频率和指定 AppWidgetProvider 类。
- 实现 AppWidgetProvider 类——定义了一些基于广播事件的基本方法,这些方法允许对 App Widget 实现编程接口。通过它,将接受到广播当 App Widgets 被更新、激活、关闭和删除的时候。
- 如何定义 Widget 的布局视图,为 App Widget 在 xml 中定义最初始的布局。
- 用于配置 App Widget 的 Activity,该 Activity 仅在需要为用户提供配置 Widget 功能时定义,它允许用户在添加 App Widget 时对它进行一些必要的设置。

接下来通过一个示例来介绍如何开发自己的 App Widget。这个示例项目的名称为 TestWidget,实现了一个 App Widget 数字时钟。这个数字时钟的效果如图 4-23 和图 4-24 所示。



图 4-23 设置 Activity

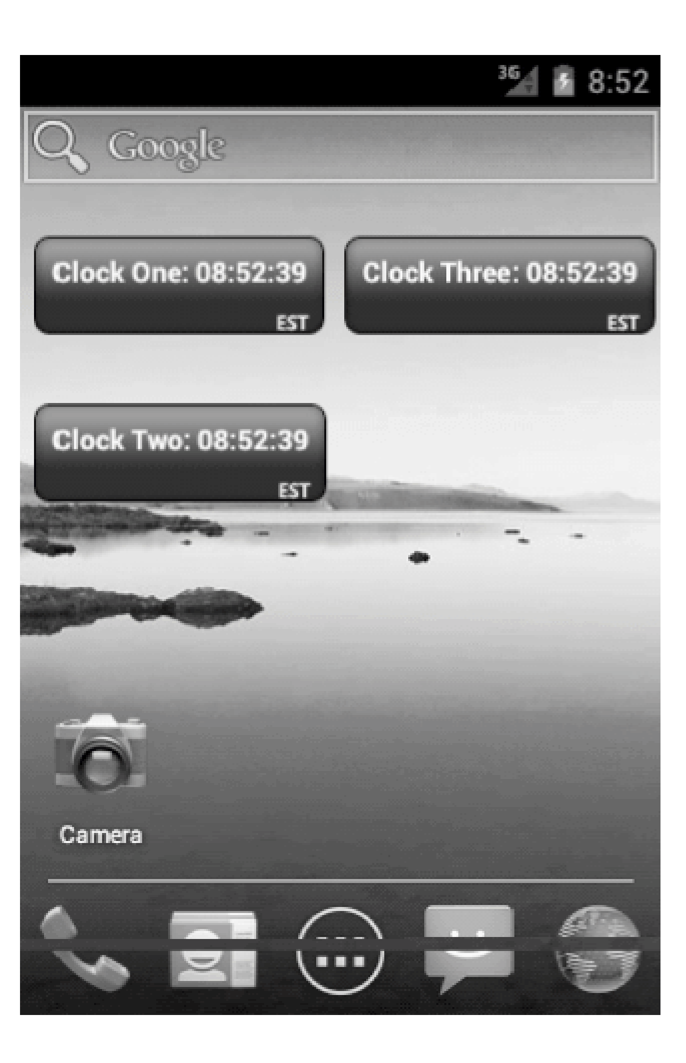


图 4-24 多个 App Widget

1. 在 Manifest 中声明 App Widget

与 Activity 和 Service 一样,首先必须在项目的 AndroidManifest.xml 文件中对 App Widget 进行声明,从而告诉系统自己提供了一个 App Widget,只有声明了的 App Widget 才能够接收到系统广播从而对自己进行更新,并使得该 App Widget 出现在系统的 Widget 列表中。声明 App Widget 的方式如下:

```
<receiver android:name = ".DigitalClockWidget" android:label = "@string/app_name">
    <intent-filter>
        <action android:name = "android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data android:name = "android.appwidget.provider"
        android:resource = "@xml/digitalclock" />
</receiver>
```

其中，第 02～04 行的 intent-filter 使得该 App Widget 能够接收 APPWIDGET_UPDATE 广播，这个广播由系统的 AppWidgetManager 发送，使得某个 App Widget 能够在需要的时候能够更新。第 05 和 06 行指定了用于描述该 App Widget Provider 的资源文件（@xml/digitalclock），android:name 属性表明这个文件是用于描述 AppWidgetProviderInfo 的。

2. 编写 appwidget-provider 元数据（metadata）

所谓元数据，就是用于描述数据的数据。在这个用于描述 App Widget 的元数据文件中，可以对 App Widget 的最小宽度（minWidth）、最小高度（minHeight）、更新周期（updatePeriodMillis）、预览图像（previewImage）、初始化布局（initialLayout）、用于配置 App Widget 的 Activity（configure）、缩放模式（resizeMode）等属性进行设置。在本例中的@xml/digitalclock文件内容为：

```
01  <?xml version="1.0" encoding="utf-8"?>
02  <appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
03      android:minWidth="146dip"
04      android:minHeight="72dip"
05      android:updatePeriodMillis="86400000"
06      android:initialLayout="@layout/digitalclock_layout"
07      android:configure="com.android.example.testwidget.AppWidgetConfigure"
08   />
```

文件为该 App Widget 设置了 5 个属性（第 03～07 行），其中第 06 行指定了该 App Widget 的布局文件（具体内容在第 3 部分），需要注意的是，为了降低电量的消耗，通常一个 App Widget 的 updatePeriodMillis 属性不要小于 1h，即一个小时最多申请一次左右的更新。在本例中设置为 86 400 000ms，即 24h。本示例实现的是一个时钟，通过在 App Widget 的 updateAppWidget()方法中启动一个按秒更新时间的线程来对显示进行更新，由于这个 updateAppWidget()方法会在 App Widget 被创建时调用一次，因此可以启动这个更新线程，但是由于设置的 updatePeriodMillis 属性为 24h，如果遇到一些事件（例如设备屏幕关闭）使得线程被终止，则可能导致该数字时钟不再走表，读者可以通过接收屏幕再次打开时（ACTION_SCREEN_ON）所产生的广播消息来重新启动这个线程。

3. 设计 App Widget 界面

前面已经提到，App Widget 的界面显示依赖于 RemoteView 类，因此可以使用一部分布局（Layout）和控件来设计 App Widget 的界面。在第 2 部分通过 meta-data 的方式为 App Widget 指定了布局文件 digitalclock_layout.xml，其具体的内容为：

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/time"
    android:textSize="14sp"
    android:textStyle="bold"
    android:textColor="#FFFFFFFF"
    android:background="@drawable/bg"
    android:layout_width="wrap_content"
```

```
        android:layout_height = "wrap_content"
    />
```

如上面的代码所示，这里没有使用复杂的布局结构，仅仅包括了一个 TextView 控件，不过在这个示例中已经足够使用了。

4. 使用 AppWidgetProvider 类管理 App Widget 事件

AppWidgetProvider 继承自 BroadcastReceiver，它用于处理与 App Widget 有关的广播事件。AppWidgetProvider 只有当 App Widget 相关的广播事件发生的时候才会接收这些广播。例如当 App Widget 更新、激活、关闭和删除的时候，将会调用 AppWidgetProvider 的如下一些方法：

- onUpdate(Context, AppWidgetManager, int[])。

这个方法按照在 updatePeriodMillis 属性中设置的数值为间隔来被调用，如果没有为 App Widget 设置用于配置的 Activity，它就会在 App Widget 第一次被创建时被调用，在这种情况下，在这个方法中就应该包括一些必要的设置和初始化 App Widget 的工作；如果设置了配置用的 Activity，则不会调用该方法。在这种情况下，就在这个 Activity 中来实现必要的设置和初始化工作。

方法的第三个参数是一个包含了 App Widget ID 的数组，用于确定哪些 App Widget 需要被更新。

- onDeleted(Context, int[])。

这个方法在 App Widget 从桌面上删除时被调用，通常在这个方法中实现对资源的释放工作。

- onEnabled(Context)。

这个方法在 App Widget 被首次创建时被调用，因为一个 App Widget 可以同时拥有多个实例，但是这个方法仅在第一个实例被创建时才会被调用。

- onDisabled(Context)。

这个方法在 App Widget 的最后一个实例从桌面上删除时被调用，这个方法与 onEnable()方法相对应，通常在这个方法中去释放在 onEnable()方法中所初始化的一些资源。

- onReceive(Context, Intent)。

这个方法在有广播时间发生时被调用，并且它的调用时间要早于上面的几个方法，通常不需要去实现这个方法，因为默认的 AppWidgetProvider 已经能够很好地对这些广播事件进行处理了。

本示例中实现的 AppWidgetProvider 中主要实现了 onUpdate()和 onDeleted()两个方法：

```
    private final static String TAG = "testAppWidget";
    private static ArrayList < Integer > activeAppWidgets = new ArrayList < Integer >();
  @Override
    public void onDeleted(Context context, int[] appWidgetIds)
```

```
{
    Log.d(TAG , "onDeleted");
    for(int i = 0; i < appWidgetIds.length; i++){
        for(int j = 0; j < activeAppWidgets.size(); j++){
            if(appWidgetIds[i] == activeAppWidgets.get(j)){
                activeAppWidgets.remove(j);
            }
        }
    }
    super.onDeleted(context, appWidgetIds);
}

@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,int[]
        appWidgetIds) {
    Log.d(TAG , "onUpdate");
    for(int i = 0; i < appWidgetIds.length; i++){
            if(!activeAppWidgets.contains(appWidgetIds[i])){
                activeAppWidgets.add(appWidgetIds[i]);
            }
    }
    final int N = appWidgetIds.length;
    for (int i=0; i<N; i++) {
        int appWidgetId = appWidgetIds[i];
        String titlePrefix = AppWidgetConfigure.loadTitlePref(context, appWidgetId);
        updateAppWidget(context, appWidgetManager, appWidgetId, titlePrefix);
    }
}
```

其中 activeAppWidgets 这个 ArrayList 类型的变量维护了当前处于活动状态的 AppWidget 的 ID 列表，具体的时间更新的操作在方法 updateAppWidget()中，在 updateAppWidget()方法中启动了一个用于更新时间的线程，每一秒更新一次时间，这个方法还将在后面实现的用于配置的 Activity 中被使用到：

```
public void updateAppWidget(final Context context,
        final AppWidgetManager appWidgetManager,
        final int appWidgetId, final String titlePrefix) {
    new Thread(){
        @Override
        public void run(){
            appWidgetDeleted = false;
            while(true){
            Time estTime = new Time();
            estTime.setToNow();
            Log.d(TAG, "update WidgetId=" + appWidgetId + " titlePrefix=" + titlePrefix);
            CharSequence text = context.getString(R.string.appwidget_text_format,
                    AppWidgetConfigure.loadTitlePref(context, appWidgetId),
                    estTime.format("%H:%M:%S"));
            RemoteViews views = new RemoteViews(context.getPackageName(),
                    R.layout.digitalclock);
            views.setTextViewText(R.id.time, text);
```

```
                appWidgetManager.updateAppWidget(appWidgetId, views);
                try{sleep(1000);}catch (Exception e) {}

                boolean stopThread = true;
                for(int j = 0; j < activeAppWidgets.size(); j++){
                    if(appWidgetId == activeAppWidgets.get(j)){
                        stopThread = false;
                    }
                }
                if(stopThread) break;
                }
            }
        }.start();
    }
```

5. 创建用于配置 App Widget 的 Activity

有时需要在创建一个 App Widget 时对它进行一些设置，这时就需要使用一个 Activity 来完成这个设置工作，在前面的 meta-data 中已经使用 android:configure 属性为 App Widget 设置了一个 Activity，即 com.android.example.testwidget.AppWidgetConfigure，本节将介绍这个 Activity 的实现。

该 Activity 的主要功能就是为一个新创建的 DigitalClock 设置一段字符串，Activity 需要在 AndroidManifest 文件中声明为一个标准的 Activity，否则 widget 在启动时抛出找不到 Activity 的异常。

```
<activity android:name="AppWidgetConfigure">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
    </intent-filter>
</activity>
```

Activity 代码如下所示：

```
public class AppWidgetConfigure extends Activity {
    private int mAppWidgetId = AppWidgetManager.INVALID_APPWIDGET_ID;
    private EditText mAppWidgetPrefix;

    public AppWidgetConfigure() {
        super();
    }

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setResult(RESULT_CANCELED);
        setContentView(R.layout.appwidget_configure);
        mAppWidgetPrefix = (EditText)findViewById(R.id.appwidget_prefix);
        Intent intent = getIntent();
```

```
        Bundle extras = intent.getExtras();
        if (extras != null) {
            mAppWidgetId = extras.getInt( AppWidgetManager.EXTRA_APPWIDGET_ID,
                    AppWidgetManager.INVALID_APPWIDGET_ID);
            System.out.println("mAppWidgetId = " + mAppWidgetId);
        }
        if (mAppWidgetId == AppWidgetManager.INVALID_APPWIDGET_ID) {
            finish();
        }
        mAppWidgetPrefix.setText(AppWidgetSharedPrefsManager
                .loadTitle(AppWidgetConfigure.this, mAppWidgetId));

        Button saveConfigure = (Button) findViewById(R.id.save_button);
        saveConfigure.setOnClickListener(new OnClickListener() {

           @Override
            public void onClick(View v) {
                final Context context = AppWidgetConfigure.this;

                String titlePrefix = mAppWidgetPrefix.getText().toString();
                AppWidgetSharedPrefsManager.saveTitle(context,
                    mAppWidgetId, titlePrefix);

                Intent resultValue = new Intent();
                resultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,mAppWidgetId);
                setResult(RESULT_OK, resultValue);
                finish();
            }
        });
    }
}
```

在 Activity 中使用了另一个类 AppWidgetSharedPrefsManager 的方法来保存每一个 App Widget 的字符串，这些字符串通过 saveTitle()方法保存到 SharedPreferences 中；通过 loadTitle()方法根据 App Widget 的 ID 来取得字符串。有关 SharedPreferences 的用法将在第 5 章进行介绍，这里仅仅给出代码。

```
public class AppWidgetSharedPrefsManager {
    private static final String PREFS_NAME = "com.android.example.AppWidgetProvider";
    private static final String PREF_PREFIX_KEY = "prefix_";

    public static void saveTitle(Context context, int appWidgetId, String text) {
        SharedPreferences.Editor prefs = context.getSharedPreferences(PREFS_NAME, 0).edit();
        prefs.putString(PREF_PREFIX_KEY + appWidgetId, text);
        prefs.commit();
    }

    public static String loadTitle(Context context, int appWidgetId) {
        SharedPreferences prefs = context.getSharedPreferences(PREFS_NAME, 0);
        String prefix = prefs.getString(PREF_PREFIX_KEY + appWidgetId, null);
        if (prefix != null) {
```

```
                return prefix;
            } else {
                return "未设置";
            }
        }
    }
```

6. 设置 App Widget 的预览效果图

这个功能是在 Android 3.0 版本之后才加入的，因此如果要使用这个预览功能，需要确定目标设备的版本在 3.0 之上。实现的方法很简单，直接截取一个图片作为预览图，然后在@xml/digitalclock 文件中添加 android:previewImage 属性：

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    ......
    android:previewImage="@drawable/preview"
/>
```

添加预览图后在 Android 的 Widgets 选项卡下的效果如图 4-25 所示。

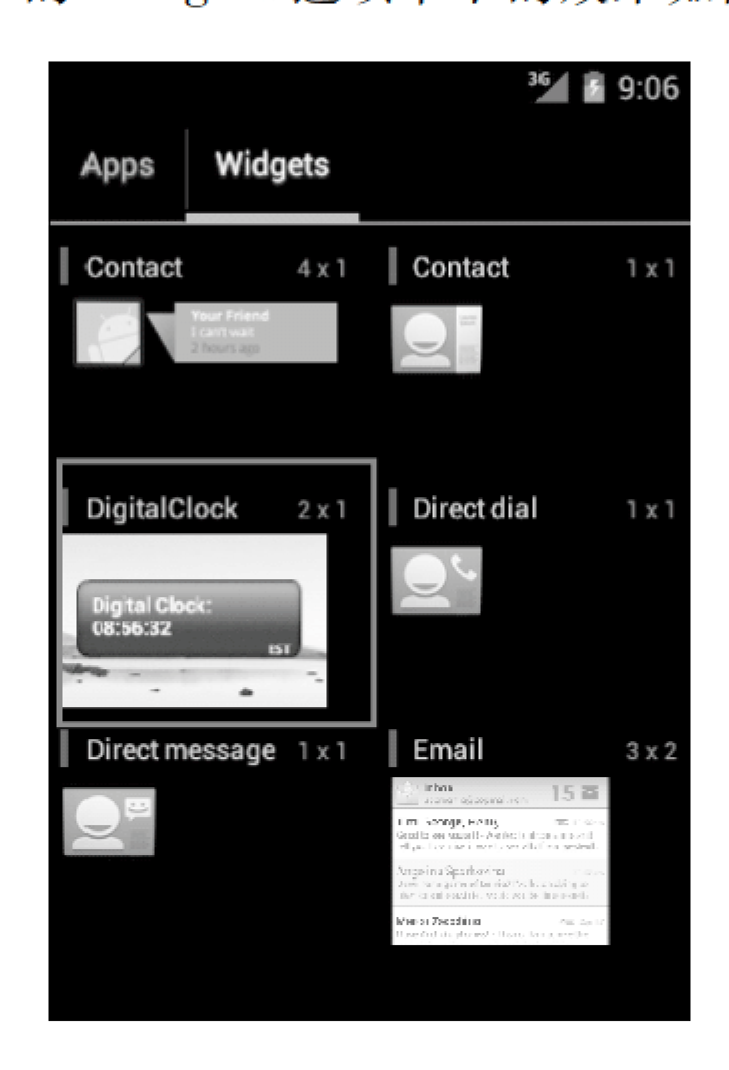


图 4-25　预览效果图

参考文献

1. Android 官方文档 Toast Notification：http://developer.android.com/guide/topics/ui/notifiers/toasts.html.
2. Android 官方文档 Status Bar Notification：http://developer.android.com/guide/topics/ui/notifiers/notifications.html.
3. Android 官方文档 Dialogs：http://developer.android.com/guide/topics/ui/dialogs.html.
4. Android 官方文档 Menus：http://developer.android.com/guide/topics/ui/menus.html.
5. Android 官方文档 App Widget：http://developer.android.com/guide/topics/appwidgets/index.html.

第5章 数据存储与共享

5.1 两种基本的数据存储方式

5.1.1 SharedPreferences

Android 中的 SharedPreferences 是用来存储简单数据的一个工具类。这个工具类与 Cookie 的概念相似，它通过用键值对的方式把简单的数据存储在应用程序的私有目录(data/data/<packagename>/shared_prefs/)下指定的 xml 文件中。SharedPreferences 是以键值对来存储应用程序的配置信息的一种方式，它只能存储基本数据类型。实际上，SharedPreferences 采用了 XML 格式将数据存储到设备中，它在 DDMS 中的 File Explorer 中的/data/data/<package name>/shares_prefs 下。以下为获取 SharedPreferences 对象的两个方法。

1. Context.getSharedPreferences(String name,int mode)

参数列表：

- name——本组件的配置文件名(如果想要与本应用程序的其他组件共享此配置文件，可以用这个名字来检索到这个配置文件)。
- mode——操作模式，默认的模式为 0 或 MODE_PRIVATE，该模式代表只自身应用程序使用，还可以使用 MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE 这两种模式，这两种模式通常用于允许外部应用程序来访问自身的 SharedPreferences，具体含义分别为：
  Context.MODE_WORLD_WRITEABLE——所有应用程序对它可写。
  Context.MODE_WORLD_READABLE——所有应用程序对它可读。
  Context.MODE_WORLD_WRITEABLE | Context.MODE_WORLD_READABLE——所有应用程序都对它可读写。

2. Activity.SharedPreferences getPreferences (int mode)

该方法可以得到一个仅限于本 Activity 使用的 SharedPreferences 对象，实际上它是直接调用了前面的 Context.getSharedPreferences(String, int)方法，并且将自己的 Activity 名作为 name 参数传入。

参数列表：

• mode——操作模式，含义与前面方法的 mode 参数一致。

SharedPreferences 提供了一种轻量级的数据存储方式，通过 edit()方法来修改存储内容，通过 commit()方法提交修改后的内容。有以下重要的使用方法：

(1) contains (String key)——检查是否已存在 key 这个关键字。

(2) edit()——为 preferences 创建编辑器 Editor，通过 Editor 可以修改 preferences 里面的数据，通过执行 commit()方法提交修改。

(3) getAll()——返回 preferences 所有的数据(Map)。

(4) getBoolean(String key, boolean defValue)——获取 Boolean 型数据。

(5) getFloat(String key, float defValue)——获取 Float 型数据。

(6) getInt(String key, int defValue)——获取 Int 型数据。

(7) getLong(String key, long defValue)——获取 Long 型数据。

(8) getString(String key, String defValue)——获取 String 型数据。

(9) registerOnSharedPreferenceChangeListener(SharedPreferences.OnSharedPreferenceChangeListener listener)——注册一个当 preference 被改变时调用的回调函数。

(10) unregisterOnSharedPreferenceChangeListener(SharedPreferences.OnSharedPreferenceChangeListener listener)——删除回调函数。

下面通过一个示例来认识这个类似于 Cookie 的 Android 简单数据存储机制。示例项目名称为 SharedPrefsDemo，该项目的主 Activity 如图 5-1 和图 5-2 所示。



图 5-1 初始状态



图 5-2 保存后状态

界面布局比较简单，即一系列的 UI 控件通过 LinearLayout 嵌套进行的布局，xml 代码就不再列出，可以到本书附带的源码(请访问清华大学出版社网站)中进行查看。下面来看一看 Activity 的实现，在 Activity 中通过单击“保存”按钮来实现对 SharedPreference 的修改和提交，鉴于读者目前可能对 Android 的代码结构尚不是十分清楚，因此此处给出完整的

代码如下（在随后的章节中，为节约篇幅，将逐渐减少给出全部代码，而是仅仅给出代码片段。代码中需要说明的地方已加注释）：

```
public class SharedPrefsDemo extends Activity {
    public static final String SETTING_INFOS = "SETTING_INFOS";
    public static final String NAME = "NAME";
    public static final String PASSWORD = "PASSWORD";
    public static final String SEX = "SEX";
    private EditText username,passwd;
    private Button save;
    private TextView status;
    private Spinner sex;
    ArrayAdapter<CharSequence> adapter;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        username = (EditText)findViewById(R.id.name);
        passwd = (EditText)findViewById(R.id.password);
        sex = (Spinner)findViewById(R.id.sex);
        save = (Button)findViewById(R.id.save);
        status = (TextView)findViewById(R.id.status_info);

        //取出已保存的值
        SharedPreferences settings = getSharedPreferences(SETTING_INFOS, 0);
        String name = settings.getString(NAME, "");
        String password = settings.getString(PASSWORD, "");
        int sex_code = settings.getInt(SEX, 0);

        save.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SharedPreferences settings = getSharedPreferences(SETTING_INFOS,0);
            //保存用户名、密码及性别
            settings.edit().putString(NAME,username.getText().toString())
                .putString(PASSWORD,passwd.getText().toString())
                .putInt(SEX, sex.getSelectedItemPosition()).commit();
                status.setText("状态提示：保存成功.可查看" +
                        "/data/data/com.android.SharedPrefsDemo" +
                        "/shared_prefs/SETTING_INFOS.xml 文件进行验证");
            }
        });

        /*为编辑框添加修改监听器*/
        username.addTextChangedListener(new TextWatcher() {
            @Override
            public void onTextChanged(CharSequence s, int start, int before, int count) {
                status.setText("状态提示：内容已重新编辑,请注意保存");
            }
            @Override
            public void beforeTextChanged(CharSequence s, int start, int count,
                    int after) {
```

```
50                  }
51                  @Override
52                  public void afterTextChanged(Editable s) {
53                  }
54              });
55
56              /* 构造性别选择列表的适配器 */
57              adapter = ArrayAdapter.createFromResource(this,
58                  R.array.sex, android.R.layout.simple_spinner_item);
59              adapter.setDropDownViewResource
60                 (android.R.layout.simple_spinner_dropdown_item);
61              /* 配置性别选择下拉列表 */
62              sex.setPrompt("请选择你的性别");
63              sex.setAdapter(adapter);
64
65              //设置值
66              username.setText(name);
67              passwd.setText(password);
68              sex.setSelection(sex_code);
69          }
70
71          protected void onStop(){
72             super.onStop();
73             passwd.setText(1);//制造异常,使进程退出
74          }
75  }
```

结合上述代码加以说明，Activity 的初始化过程是：在 onCreate 方法中使用 getSharedPreferences 取得 SharedPreferences 的对象 settings(第 22 行)，然后使用 getString 和 getInt 来取得其中保存的值(第 23～25 行)，最后使用 setText 和 setSelection 将保存的值赋值给编辑框和下拉列表(第 66～68 行)。

当单击"保存"按钮时，将触发绑定在该按钮上的点击事件监听器，即执行代码第 27～39 行，会首先通过 getSharedPreferences()方法得到 settings，然后调用 edit()方法得到编辑器 Editor，使用 Editor 的 putString 和 putInt 将编辑框及下拉列表的值进行修改，最后使用 commit()方法将数据提交保存。SharedPreferences 以 xml 文件保存需要保存的值。更重要的是，SharedPreferences 只能由所属 package 的应用程序使用，而不能被其他应用程序使用，从而提高了安全性。

当程序退出时，onStop()方法被调用，这里为了使程序完全退出，制造了一个异常(代码第 73 行)，因为由于 Activity 的生命周期是由系统管理，在使用 Back 键关闭一个 Activity 时，该 Activity 不会完全被销毁，而是驻留在内存中，因此在本示例中如果不完全退出应用程序进程，会导致删除了存储 SharedPreferences 信息的 xml 文件后再次打开应用程序时，只要 Activity 没有被销毁即 onDestroy 方法没有被调用，编辑框和下拉列表的数据会由于直接从内容中恢复而仍然存在，从而影响示例的演示，因此采用了造成异常来关闭应用程序进程。同时，为了使异常退出时不弹出"Force Close 窗口"，这里通过继承 Application 类实现 MyApp 类并重写异常处理方法来解决这个问题。经过如此处理之后，该示例就能够更好地解释 SharedPreferences 的使用原理：SharedPrefs 以 xml 文件的形式来存储少量的用

户数据，该 xml 文件存放在系统的/data/data/＜package name＞/shared_prefs/路径下，可以通过编辑或删除这个文件来进行验证。MyApp 代码如下：

```
public class MyApp extends Application implements UncaughtExceptionHandler {
    @Override
    public void uncaughtException(Thread thread, Throwable ex) {
        Log.i("duanhong", "stop!!!!");
        System.exit(0);
    }
    @Override
    public void onCreate() {
        super.onCreate();
        //修改异常处理器，从而防止 Force Close 对话框弹出
        Thread.setDefaultUncaughtExceptionHandler(this);
    }
}
```

5.1.2　文件存储：File

虽然 5.1.1 节介绍的 SharedPreferences 可以非常方便地存储数据，但是这种方式只适用于比较少量的数据，在大量数据需要存储时，可以借助于文件存储的功能。借助于 Java 文件 I/O 类，使用 FileInputStream 和 FileOutputStream 类来读取和写入文件，典型代码如下：

```
String FILE_NAME = "filename.txt";    //确定要操作文件的文件名
FileOutputStream fos = openFileOutput(FILE_NAME,Context.MODE_PRIVATE);    //输出流
FileInputStream fis = openFileInput(FILE_NAME);    //输入流
```

使用文件输入输出流时需要知道如下几点：

(1) 如果在创建 FileOutputStream 时指定的文件不存在，系统会自动创建这个文件。

(2) 默认的写入操作会覆盖源文件的内容，如果想要把新写入的内容附加在原文件的内容之后，可以指定模式为 Context.MODE_APPEND。

(3) 默认情况下，使用 openFileOutput 方法打开的文件只能被其调用的应用程序使用，其他应用程序将无法读取这个文件。

(4) 如果需要在不同的应用程序中共享数据，可以使用 ContentProvider，这个方法将在 5.3 节中介绍。

有关 File 相关使用方法的示例项目名称为 FileIODemo，这里简单地对其进行一下分析。该示例的运行效果如图 5-3～图 5-6 所示。

该示例虽然实现的功能比较简单，但是却包含了许多前面所学习过的知识点。首先从布局上来说，根节点的布局是一个垂直方向排列的 LinearLayout，依次包含了一个 TextView、一个 EditText、一个水平方向排列的 LinearLayout、一个 TextView，最后是一个 ListView。其中的 EditText 还用到了最小显示行数及最大显示行数(超过最大显示行数将出现滚动条)、垂直方向的滚动条、提示文字等属性，横向 LinearLayout 使用了 layout_weight 即权重属性使得 3 个按钮的宽度按合理的比例(2∶2∶1)得到分配；另外还在新建

文件和保存编辑时使用了对话框进行操作提示。ListView 的显示采用了继承 ListActivity 的方法来实现，ListActivity 是 Android 提供的专用于在 Activity 中显示一个 ListView 的 Activity 基类，它的默认布局是存在于屏幕中央的一个 ListView，开发者也可以使用自己的 layout 布局，但是在 layout 中必须包含一个 id 为@android:id/list 的 ListView 控件，使用此种类型的 Activity 类显示 Listview 就很简单了，只需要为自身设置好适配器 Adapter 即可。

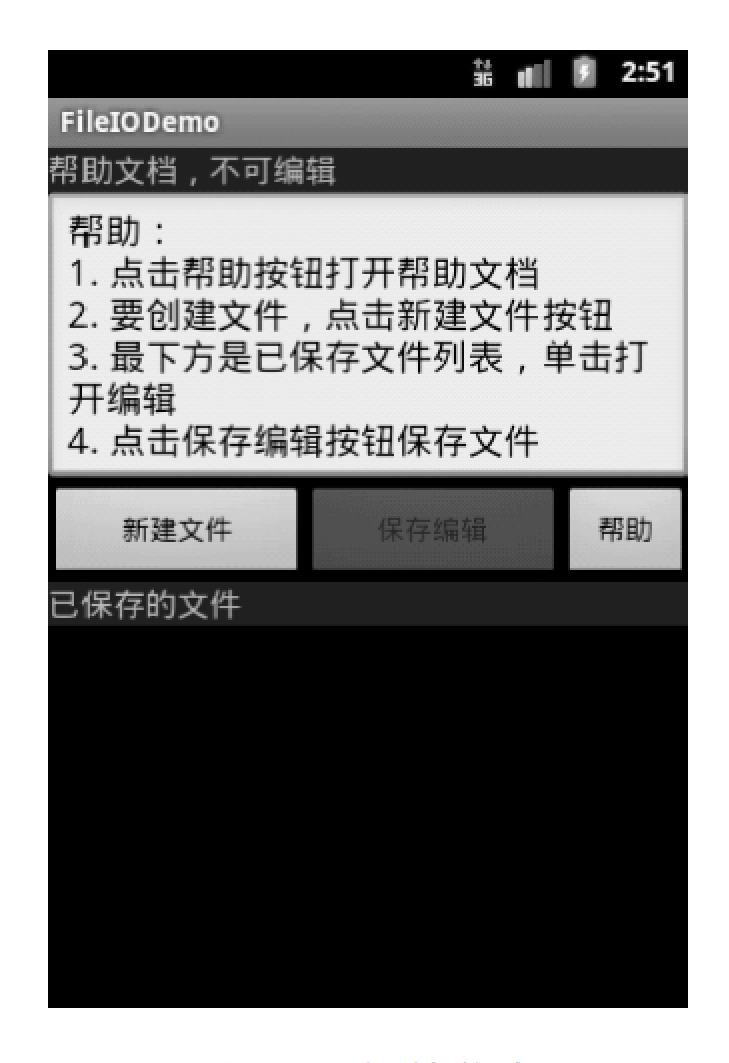


图 5-3　初始状态



图 5-4　新建文件



图 5-5　保存成功



图 5-6　文件列表

该 Activity 主要包括了如下几个方法。

1. savefile()

用于保存文件。保存文件的过程就是先使用 FileOutputStream 创建输出流，然后获取

待写入到文件中的数据并写入文件中。FileOutputStream 写文件的方法是使用 write()方法,使用 flush()方法保证输出流写入完成,最后使用 close()方法关闭输出流,文件保存完毕。

```
protected void savefile() throws IOException {
FileOutputStream fos = new FileOutputStream(mTextFile);
    fos.write(et.getText().toString().getBytes());
    fos.flush();//确保输出完毕
fos.close();
}
```

2. helpdoc()

```
01 private void helpdoc() throws IOException{
02     save.setClickable(false);
03     save.setEnabled(false);
04     tw.setText("帮助文档,不可编辑");
05     String myString = null;
06     InputStream is = getApplicationContext().getContentResolver()
07         .openInputStream(Uri.parse("android.resource://" +
08                 "com.android.example.fileiodemo/" + R.raw.help));
09     BufferedInputStream bis = new BufferedInputStream(is);
10     ByteArrayBuffer baf = new ByteArrayBuffer(8192);
11     int current = 0;
12     while((current = bis.read()) != -1) {
13         baf.append((byte)current);
14     }
15     myString = new String(baf.toByteArray(),"GBK");
16     et.setText(myString);
17 }
```

用于显示该程序的帮助文档,帮助文档对于用户来说是个不可或缺的部分,在此项目中仅作简单的示例,任何一款好的应用程序都有着详细而清楚的帮助文档,能帮助用户快速掌握程序的相关用法。帮助文档的实现代码很简单,需要注意的有两点:第一,由于帮助文档需要在应用程序安装的时候一并装载到设备中,因此选择将帮助文档存放于 res/raw 目录下,在代码中访问该目录下文件的方式为使用如下所示的 URI,即前面代码的第 07 行和第 08 行:

```
"android.resource://com.android.example.fileiodemo/" + R.raw.help
```

即在对应文件的 id 前面加上一个用于表示 raw 文件地址的 URI 前缀;第二,由于帮助文档是由 PC 上编辑完成之后纳入到项目目录下,因此可能会涉及编码的问题,对于中文显示乱码的问题,就需要将输出字符编码类型设定为 GBK,即前面代码的第 15 行:

```
myString = new String(baf.toByteArray(),"GBK");
```

3. readfile()

用于打开文件。该例中文件打开需要的步骤是使用 FileInputStream 得到待打开文件的输入流,然后从输入流中读出所包含的数据内容并显示到文本框中即可。

```
private void readfile(File file) throws IOException{
    mTextFile = file;
    tw.setText("正在编辑文件" + mTextFile.getName());
    if(!mTextFile.exists()){
      Log.v("duanhong", "创建文件");
      if(!mTextFile.createNewFile()){
        Log.v("duanhong","创建文件失败");
        return;
      }
    }
    String myString = null;
    InputStream is = new FileInputStream(mTextFile);//创建写入流
    BufferedInputStream bis = new BufferedInputStream(is);
    ByteArrayBuffer baf = new ByteArrayBuffer(8192);
    int current = 0;
    while((current = bis.read()) != -1) {
        baf.append((byte)current);
    }
    myString = new String(baf.toByteArray());
    et.setText(myString);
}
```

4. 重写 onListItemClick()方法

用于相应 ListView 内容的点击事件。

```
@Override
/* 点击列表中某项时,打开被点击的文件 */
protected void onListItemClick(ListView l,
        View v, int position, long id)
{
    /* 得到被点击的文件 */
    mTextFile = new File(mTextFilePath.getAbsolutePath()
            + File.separator + mTextFileList.get(position));
    /* 显示文件内容 */
    try {
        readfile(mTextFile);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    save.setClickable(true);
    save.setEnabled(true);
}
```

5. textFileList()

用于列出已经保存过的文件列表,供用户浏览、编辑。

```
/* 已保存文件列表 */
public void textFileList()
{
    //取得指定位置的文件设置显示到文件列表
    File home = mTextFilePath;
    if (home.listFiles(new TextFilter()).length > 0)
    {
        for (File file : home.listFiles(new TextFilter()))
        {
            mTextFileList.add(file.getName());
        }
        ArrayAdapter<String> textFileList =
                new ArrayAdapter<String>(FileIODemoActivity.this,
                        R.layout.list, mTextFileList);
        setListAdapter(textFileList);
    }
}
```

6. 重写 onCreateDialog()方法

用于弹出保存提示及新建文件提示对话框。

```
@Override
protected Dialog onCreateDialog(int id) {
    switch (id) {
        case R.id.dialog_save_success: {
          AlertDialog.Builder builder = new AlertDialog.Builder(this)
          .setTitle("操作提示")
          .setIcon(android.R.drawable.ic_dialog_alert)
          .setMessage("保存成功")
          .setOnCancelListener(null)
          .setPositiveButton(android.R.string.ok, null);
          return builder.create();
        }
        case R.id.dialog_create_success: {
          AlertDialog.Builder builder = new AlertDialog.Builder(this)
          .setTitle("操作提示")
          .setIcon(android.R.drawable.ic_dialog_alert)
          .setMessage("新建文件" + mTextFile.getName() + "成功")
          .setOnCancelListener(null)
          .setPositiveButton(android.R.string.ok, null);
          return builder.create();
        }
    }
    return super.onCreateDialog(id);
}
```

除了实现上述方法外,另外还实现了一个用于过滤文件类型的内部类 TextFilter,由于

该示例设计到的 File 都是文本文件，因此使用 TextFilter 过滤出 txt 类型的文件，以便显示到列表视图中。

```
//过滤文件类型
class TextFilter implements FilenameFilter
{
    public boolean accept(File dir, String name)
    {
        return (name.endsWith(".txt"));
    }
}
```

5.2 使用 SQLite 数据库存取数据

5.2.1 SQLite 简介

SQLite 是一款开源的轻量级嵌入式关系型数据库。它在 2000 年由 D. Richard Hipp 发布，支持 Java、Net、PHP、Ruby、Python、Perl、C 等几乎所有的现代编程语言，并且支持 Windows、Linux、Unix、Mac OS、Android、iOS 等几乎所有的主流操作系统平台。

SQLite 支持多数 SQL92 标准，可以在所有主要的操作系统上运行，并且支持大多数计算机语言。SQLite 非常健壮。其创建者保守地估计 SQLite 可以处理每天负载多达 10 000 次点击率的 Web 站点，并且 SQLite 有时候可以处理 10 倍于上述数字的负载。

SQLite 对 SQL92 标准的支持包括索引、限制、触发和查看。SQLite 不支持外键限制，但支持原子的、一致的、独立和持久（ACID）的事务。这意味着事务是原子的，因为它们要么完全执行，要么根本不执行；事务也是一致的，因为在不一致的状态中，该数据库从未被保留。事务还是独立的，所以，如果在同一时间在同一数据库上有两个执行操作的事务，那么这两个事务是互不干扰的；而且事务是持久性的，所以，该数据库能够在崩溃和断电时幸免于难，不会丢失数据或损坏。SQLite 通过数据库级上的独占性和共享锁定来实现独立事务处理。这意味着当多个进程和线程可在同一时间从同一数据库读取数据时，只有一个可以写入数据。在某个进程或线程向数据库执行写入操作之前，必须获得独占锁定。在发出独占锁定后，其他的读或写操作将不会再发生。

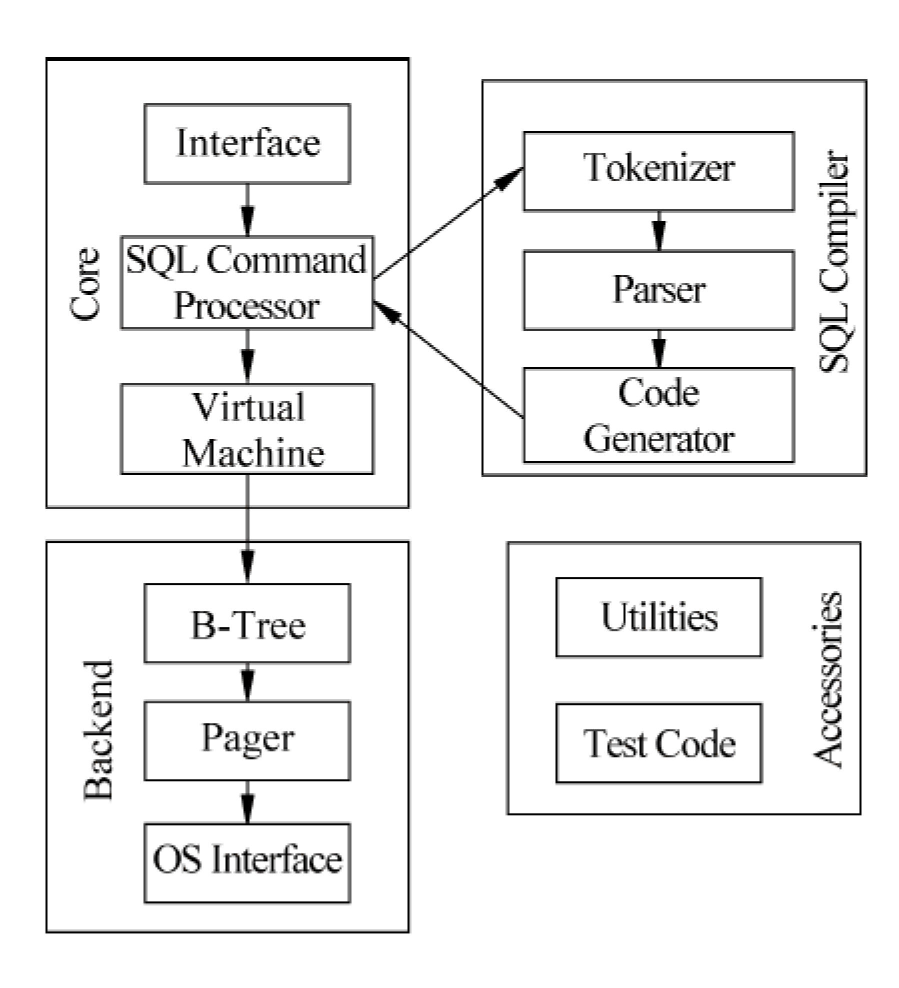


图 5-7　SQLite 内部结构

SQLite 的内部结构如图 5-7 所示。

由于资源占用少、性能良好和零管理成本，嵌入式数据库 SQLite 有了它的用武之地，它将为那些以前无法提供用作持久数据的后端的数据库的应用程序提供了高效的性能。现在，没有必要使用文本文件来实现持久存储。SQLite 之类的嵌入式

数据库的易于使用性可以加快应用程序的开发，并使得小型应用程序能够完全支持复杂的SQL。这一点对于小型设备空间的应用程序来说尤其重要。

SQLite 被广泛应用在苹果、Adobe、Google 的各项产品中。日常生活中所使用到的诸多应用程序中也不难发现 SQLite 的影子，如果读者的计算机中装有迅雷，请打开迅雷安装目录，搜索 SQLite3.dll 就能够发现该文件，从名称上基本能够说明迅雷使用了 SQLite 作为其数据库；还有金山词霸，它的安装目录中也能够发现 SQLite.dll 的存在。是的，SQLite 早就广泛地应用在日常生活中人们所接触的各种产品中了。在 Android 中还内置了完整支持的 SQLite 数据库。

SQLite 数据库具有如下的一系列特性：

- 遵守 ACID。
- 零配置——无须安装和管理配置。
- 存储在单一磁盘文件中的一个完整的数据库。
- 数据库文件可以在不同字节顺序的机器间自由的共享。
- 支持数据库大小至 2TB。
- 足够小，约 3 万行 C 代码，250KB。
- 在进行大部分普通的数据库操作时，它的速度比其他一些流行的数据库要快。
- 简单的 API。
- 包含 TCL 绑定，同时通过 Wrapper 支持其他语言的绑定。
- 良好注释的源代码，并且有着 90%以上的测试覆盖率。
- 独立——没有额外依赖。
- Source 完全的 Open，可以用于任何用途，包括出售它。
- 支持多种开发语言：C、PHP、Perl、Java、ASP.NET 和 Python。

5.2.2 实现 SQLite 数据库访问器

为了实现对 SQLite 数据库的便捷访问，可以通过实现一个数据库访问器类来封装对数据库的基本操作并为上层提供接口，在对数据库进行操作的时候有两种可供选择的方法：一种是直接使用 SQL 语句进行操作，SQLiteDatabase 类提供了 execSQL()方法用于执行给定的 SQL 语句；另一种方法就是使用由 SQLiteDatabase 类所实现的一系列数据库操作方法，例如用于插入的 insert()方法、用于替换表项的 replace()方法、用于删除表项的 delete()方法等，还提供了用于返回查询结果 Cursor 的 rawQuey()系列的方法，借助于这些方法也可以很方便地操作数据库。在本例中对上述两种方法都有部分使用到。DBHelper 辅助类中包括了创建数据库，创建表以及使用数据库的 insert、delete、update、select 方法，所有与数据库相关的操作都放在了这个辅助类当中，因此 SQLiteDemoActivity 就可以借助于 DBHelper 来访问数据库。下面具体的分析数据库操作辅助类 DBHelper 的代码实现。DBHelper 提供了如下几个方法和接口。

1. void CreateTable()

该方法用于创建所需要的数据库表，此处使用的是 SQLite 数据库类的 execSQL 方法来执行 SQL 语句来完成数据库表的创建，如代码所示，如果 SQL 语句成功执行，在指定的

表不存在的前提下，将会创建一个含有_id、fileName、description 3 个字段的表：

```
private void CreateTable() {
//使用 execSQL 方法执行 SQL 语句完成数据库表创建
try {
        db.execSQL("CREATE TABLE IF NOT EXISTS " + table_name + "(" +
                "_id INTEGER PRIMARY KEY AUTOINCREMENT  NOT NULL,"
                + "fileName VARCHAR, description VARCHAR"
                + ");");
        Log.v(TAG, "Create Table files ok");
    } catch (Exception e) {
        Log.v(TAG, e.toString());
    }
}
```

2. void initDatabase()

该方法用于初始化表中的数据，此处为了便于测试，向表中插入了一条记录。

```
//初始化表,使用 SQLiteDatabase 提供的 insert 方法插入一行数据
public void initDatabase(){
    ContentValues cv = new ContentValues();    //数据集,表示一行数据
    cv.put("fileName", "Test");
    cv.put("description", "初始化测试项");
    db.insert(table_name, "", cv);
}
```

3. boolean insert (String filename, String description)

参数列表：

- filename——待插入条目的名称。
- description——待插入条目的描述。

返回值：

- boolean——插入条目操作是否成功，成功则返回 true，不成功则返回 false。

该方法用于向数据库中插入一条数据，该示例的功能是模拟一个文件列表的功能，因此待插入的项需要提供其名称和描述。

```
/**
 * @param filename 欲插入条目的名称
 * @param description 欲插入条目的描述
 * @return 条目是否插入成功
 */
public boolean insert(String filename, String description){
    String sql = "";
    try{
     //打开数据库供后续操作使用
     db = context.openOrCreateDatabase(name, Context.MODE_PRIVATE,null);
     sql = "insert into files values(null,'" + filename + "','" + description +"')";
        db.execSQL(sql);
```

```
            Log.v(TAG,"insert Table files ok");
            return true;
        }catch(Exception e){
            Log.v(TAG,"insert Table files err ,sql: " + sql);
            return false;
        }
    }
```

4. boolean delete(int fileId)

参数列表：

- fileId——欲删除条目的 id。

返回值：

- boolean——插入条目操作是否成功，成功则返回 true，不成功则返回 false。

该方法将会根据传入的 fileId 参数来删除数据库表中对应的条目。

```
/**
 * @param fileId 欲删除的条目 id
 * @return 条目是否删除成功
 */
public boolean delete(int fileId){
    String sql = "";
    try{
        db = context.openOrCreateDatabase(name, Context.MODE_PRIVATE,null);
        sql = "delete from files where _id = " + fileId;
        db.execSQL(sql);
        Log.v(TAG,"delete item ok");
        return true;

    }catch(Exception e){
        Log.v(TAG,"delete item err ,sql: " + sql);
        return false;
    }
}
```

5. boolean update(int fileId, String filename, String description)

参数列表：

- fileId——欲更新条目的 id。
- filename——将条目的 filename 属性更新成为相应的值。
- description——将条目的 description 属性更新成为相应的值。

返回值

- boolean——插入条目操作是否成功，成功则返回 true，不成功则返回 false。

该方法的作用是根据传入的参数，修改指定 fileId 所对应的 filename 或 description 属性为新的值。

```
/**
 * @param fileId 欲修改的条目 id
 * @param filename 欲修改条目名称的目标内容
 * @param description 欲修改条目的目标描述
 * @return 条目是否修改成功
 */
public boolean update(int fileId, String filename, String description){
    String sql = "";
    try{
        db = context.openOrCreateDatabase(name, Context.MODE_PRIVATE,null);
        sql = "update files set fileName = '" + filename + "',description = '"
            + description + "' where _id = " + fileId;
        db.execSQL(sql);
        Log.v(TAG,"update Table files ok");
        return true;

    }catch(Exception e){
        Log.v(TAG,"update Table files err ,sql: " + sql);
        return false;
    }
}
```

6. Cursor select(int fileId)

参数列表:

- fileId——欲查询条目的 id。

返回值:

- Cursor——可以根据其获取到查询结果的 Cursor 对象。

该方法的作用是根据经参数传入的条目 id 查询出对应的结果,并将该结果保存在 Cursor 对象中返回,使得调用该方法的主体能够获取到欲查询条目的相关信息,由于该方法需要返回值,因此不能够再使用 execSQL 执行 SQL 语句的方式进行查询,而是使用 SQLiteDatabase 提供的 query()方法进行,如下面代码的第 08～10 行。

```
01  /**
02   * @param fileId 欲查询的条目 id
03   * @return 欲查询条目的 Cursor,使用该 Cursor 可得到条目各属性内容
04   */
05  public Cursor select(int fileId){
06      String sql = "_id = " + fileId;
07      db = context.openOrCreateDatabase(name, Context.MODE_PRIVATE,null);
08      Cursor cur = db.query(table_name,
09          new String[]{"_id","fileName","description"},
10          sql, null, null, null, null);
11      return cur;
12  }
```

7. Cursor loadAll()

返回值:

- Cursor——可以根据其获取到查询结果的 Cursor 对象。

该方法用于获取一个指代了数据库中所有条目的 Cursor 对象，这个 Cursor 对象通常的用途是刷新视图显示。

```
public Cursor loadAll(){//返回可得到数据库所有表项的 Cursor
  db = context.openOrCreateDatabase(name, Context.MODE_PRIVATE,null);
    Cursor cur = db.query(table_name,
          new String[]{"_id","fileName","description"},
          null, null, null, null, null);
    return cur;
}
```

8. DBHelper(Context context, String name, CursorFactory factory,int version)

参数列表：

- context——应用环境上下文，可以在实例化该对象的 Activity 中获取。
- name——该 DBHelper 所对应的数据库名。
- factory——用于返回 Cursor 的工厂类，此处未直接使用该参数。
- version——数据库的版本号，该版本号配合 onUpgrade()方法和 onDowngrade()方法使用。

该方法是 DBHelper 类的构造方法，由于 DBHelper 继承自 SQLiteOpenHelper 类，因此此处保留了 SQLiteOpenHelper 构造方法的参数列表，实际上此处仅仅使用该构造函数的前两个参数，即传入应用环境上下文以及数据库名，根据这两个参数来使用数据库。

```
public DBHelper(Context context, String name, CursorFactory factory, int version) {
    super(context, name, factory, version);
    this.context = context;
    this.name = name;
    db = context.openOrCreateDatabase(name, Context.MODE_PRIVATE, null);
    drop_table();
    CreateTable();
}
```

从上面的代码中可以看到，SQLiteDatabase 的对象是由 context. openOrCreateDatabase (name, Context. MODE_PRIVATE, null)方法返回的，即数据库的创建工作实际上是由应用程序来负责创建的。

5.2.3 SQLite 示例

在前一节中已经实现了用于操作数据库的一些必要的方法，利用上述的一些方法就可以对数据库进行一些简单的操作了。本节将利用前面所实现的一系列方法，来实现一个简单的 SQLite 示例。首先给出示例的一些截图，如图 5-8～图 5-13 所示，下面将结合这些截图来说明其实现的功能。

该示例的功能是在可视化的界面中来操作数据库，首先在一个列表视图中显示出当前数据库的内容，并且提供“增加记录”、“删除记录”、“修改记录”和“查询记录”4 个功能。





图 5-8 初始状态



图 5-9 插入新的项



图 5-10 删除项错误



图 5-11 删除条目 3 成功




图 5-12 修改条目 4 成功



图 5-13 查询条目 5

- 示例应用程序在启动时将会检查并创建数据库(通过 DBHelper 构造方法)，然后做数据库的初始化工作(见图 5-8)。
- 插入新的条目：界面中提供了 3 个可编辑的文本框用于完成输入和输出工作。插入条目时，只有右边两个文本框是有意义的，在这两个文本框中分别输入需要插入条目的“条目名称”和“条目描述”信息，然后单击“增加记录”按钮即可插入新的条目(见图 5-9)。
- 删除一个条目：当需要进行删除条目操作时，只有左边第一个文本框是有意义的，并且只有文本框中的输入为数字时，才能够进行正常的删除操作，输入所需要的信息后单击“删除记录”按钮即可执行删除操作。如果输入不为数字，则会出现提示消息“请输入纯数字的 id 号”(见图 5-10)，当输入正确时，并且数据库中已经含有指定

id 的条目，则删除操作成功，数据库中对应条目被删除（见图 5-11）。

- 修改一个条目：当需要进行修改条目内容的操作时，3 个文本框都是有意义的，第一个文本框用于输入需要修改的条目的 id，后面两个文本框则用于输入更新内容（见图 5-12）。
- 查询数据库：可以通过 id 值查询数据库，在第一个文本框中输入欲查询条目的 id，单击“查询条目”按钮，在后面的两个文本框中将会返回查询的结果（见图 5-13）。

上面介绍了示例所具备的功能，接下来通过代码来了解这些功能的具体实现，首先是实现初始化工作的代码：

```
//实例化辅助类,初始化数据
helper = new DBHelper(this,db_name,null,2);
helper.initDatabase();
refresh();
```

首先使用 DBHelper 的构造方法创建了一个 DBHelper 的对象，然后该对象调用自身的 initDatabase()方法初始化数据库，向数据库中插入一个新的条目，最后调用了 refresh()方法刷新数据库对应的视图显示。refresh()是通过 DBHelper 的 loadAll()方法来获取能够访问所有表项的 Cursor，然后根据 Cursor 创建新的 ListAdapter，从而达到刷新视图的效果，其代码如下：

```
//刷新数据库内容显示
public void refresh(){
    Cursor cur = helper.loadAll();
    //将 cursor 对象交给系统管理,使 cursor 与 Activity 生命周期同步
    startManagingCursor(cur);
    ListAdapter la = new SimpleCursorAdapter(this,R.layout.list_item,
        cur,new String[]{"_id", "fileName", "description"},
        new int[]{R.id.item_fileid, R.id.item_filename,
        R.id.item_description});
    items.setAdapter(la);
    items.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
    helper.close();
}
```

其余的 4 个功能实现的方式为：为视图添加 4 个代表对应功能的按钮，然后为这 4 个按钮绑定同一个点击事件监听器（OnClickListener），该事件监听器通过分析按键的 id 来采取正确的操作，监听器代码如下：

```
OnClickListener ocl = new OnClickListener(){
    public void onClick(View v) {
        int id;
        switch(v.getId()){
        case R.id.add:
            helper.insert(filename.getText().toString(),
                    description.getText().toString());
            refresh();
            break;
        case R.id.delete:
```

```
11                try {
12                    id = Integer.parseInt(fileid.getText().toString());
13                }catch (NumberFormatException e) {
14                    answer.setText("请输入纯数字的 id 号");
15                    break;
16                }
17                helper.delete(id);
18                refresh();
19                break;
20            case R.id.modify:
21                try {
22                    id = Integer.parseInt(fileid.getText().toString());
23                }catch (NumberFormatException e) {
24                    answer.setText("请输入纯数字的 id 号");
25                    break;
26                }
27                helper.update(id, filename.getText().toString(),
28                        description.getText().toString());
29                refresh();
30                break;
31            case R.id.query:
32                try {
33                    id = Integer.parseInt(fileid.getText().toString());
34                }catch (NumberFormatException e) {
35                    answer.setText("请输入纯数字的 id 号");
36                    break;
37                }
38            Cursor cur = helper.select(id);
39                cur.moveToFirst();
40                filename.setText(cur.getString(1));
41                description.setText(cur.getString(2));
42                break;
43            }
44        }
45    };
```

从上面的代码中可以注意到，监听器在响应除了查询操作之外的 3 种操作时都会调用一次 refresh()方法（代码第 08 行、第 18 行和第 29 行），通过这种方式来使得数据库视图得到及时的更新；监听器在响应除了插入操作之外的 3 种操作时都会对 id 输入框的内容进行判定，通过这种方式来保证输入的合法性（代码第 11～16 行、第 21～26 行、第 32～37 行）。

实现了监听器之后，再分别为 4 个按钮注册该监听器即可。

```
//按钮注册监听器
add.setOnClickListener(ocl);
delete.setOnClickListener(ocl);
modify.setOnClickListener(ocl);
query.setOnClickListener(ocl);
```

5.3 Content Provider

5.3.1 Content Provider 简介

通常对于每个应用程序来说，其自身所维护的数据是仅仅受限于自身使用的，其他应用程序对这部分数据不感兴趣或者没有权限对这部分数据进行访问。例如，一个浏览器应用程序会保存网页历史记录、缓存、表单和密码等数据，而其他应用程序则对这部分数据的使用方法不清楚，因此不能正确地使用它们；一个电子书阅读程序则会按一定的方式保存电子书资源，这些资源的使用方法只需要阅读器自身访问而不需要能够使其他应用程序访问。

Android 已经很好地对这种数据访问情况进行了限制。在 Android 平台下，应用程序的数据都做了严格的绑定保护，应用程序不能够直接跨过 package 去读写其他应用程序的数据。

但是在实际使用过程中又经常会遇到需要应用程序之间共享数据的情况，例如，属于 contacts 应用程序的联系人信息通常需要将这些数据提供给拨号程序、短信息程序、社交网络应用等使用；属于多媒体管理器程序的多媒体文件信息需要提供给音乐播放器、视频播放器等应用使用。在这样的情况下就需要掌握数据的应用程序给外界提供一组用于访问数据的接口。

Android 所提供的 content provider API 就是用于方便地实现应用程序之间共享数据接口的工具，通过 content provider API，所有的应用程序都可以向 OS 发出请求以完成数据查询或其他操作，而具体需要访问的数据库通过 URI 来进行标识。在 Android 中，使用 URI(Uniform Resource Identifier，统一资源标识符)来定位文件和数据资源。相比常见的与之容易混淆的是 URL(Uniform Resource Locator，统一资源定位器)，URL 是用于标识资源的物理位置，相当于文件的路径；而 URI 则是标识资源的逻辑位置，并不提供资源的具体位置。例如 Android 通讯录中的数据，如果使用 URL 来标识，可能会是一个很复杂的定位结构，并且一旦文件的存储路径改变，URL 也必须随之改动；而对于 URI，可以用诸如 content://contract/people 这样的逻辑地址来标识，对于用户来说，这种方式不需要关心文件的具体位置，即使文件位置改动也不需要变化，后台程序中 URI 到具体位置的映射通过程序员来进行维护。

ContentProvider 是应用程序私有数据对外的接口，程序通过 ContentProvider 访问数据时不需要关心数据具体的存储及访问过程，这样的方式既提高了数据的访问效率，同时也保护了数据。Activity 类中有一个继承自 ContentWapper 的 getContentResolver()无参数方法，该方法返回一个 ContentResolver 对象，通过调用其 query、insert、update、delete 方法访问数据。这几个方法的第一个参数均为 URI，用来标识需要访问的资源或数据库。

ContentProvider URI 固定的形式如下，以联系人应用程序为例：

content://contract/people/001

A B C D

A——schema(模式)，类似于 URL 中的 http://、ftp://等。

B——authority(授权)，是一种唯一的标识符，可以简单地理解为代表某个数据库。

C——具体的资源类型，可以理解为数据库表名。

D——ID 号，用于指定一条数据，可以理解为数据库中的某一行的 id。

ContentResolver 是用于访问通过 ContentProvider 获取的其他应用程序所共享的数据的类，简单地说就是一个提供数据，一个处理数据。其中 ContentProvider 负责：

（1）组织应用程序的数据。

（2）向其他应用程序提供数据。

ContentResolver 负责：

（1）获取 ContentProvider 提供的数据。

（2）修改/添加/删除更新数据等。

ContentProvider 通过如下方式向外界提供数据：应用程序可以通过实现一个 ContentProvider 的抽象接口将自己的数据对外开放，ContentProviders 以类似数据库中表的方式将数据暴露出来，因此可以将 ContentProvider 理解为数据库，外界获取其提供的数据的方式与从数据库中获取数据的操作基本一样，只不过这里采用 URI 来表示要访问的数据库。而如何从 URI 中识别出要访问的是哪个数据库的工作由 Android 底层来完成。下面简要说明 ContentProvider 向外界提供数据操作的接口：

```
query(Uri, String[], String, String[], String)
insert(Uri, ContentValues)
update(Uri, ContentValues, String, String[])
delete(Uri, String, String[])
```

这些操作与数据库的操作基本一样，此处不再赘述。

ContentProvider 组织数据主要包括：存储数据、读取数据、以数据库的方式暴露数据。数据的存储需要根据设计的需求，选择合适的存储结构，首选数据库，当然也可以选择本地其他文件，甚至可以是网络上的数据。数据的读取，以数据库的方式暴露数据这就要求，无论数据是如何存储的，数据最后必须以数据的方式访问。

除了上面的内容，ContentProvider 有下面两个问题值得留意：

（1）ContentProvider 是什么时候创建的？是谁创建的？访问某个应用程序共享的数据，是否需要启动这个应用程序？这个问题在 Android SDK 中没有明确说明，ContentProvider 是 Android 在系统启动时就创建了。访问某个应用程序的共享数据不需要启动那个应用程序，因为数据已经通过在该应用程序的 AndroidManifest.xml 中定义的<provider>元素方式对外部开放了，其他应用程序只需要按照约定使用数据即可。

（2）若多个程序同时通过 ContentResolver 访问一个 ContentProvider，会不会导致类似数据库的“脏数据”？这个问题一方面需要数据库访问的同步，尤其是数据写入的同步，在 AndroidManifest.xml 中定义 ContentProvider 的时候，需要考虑是<provider>元素 multiprocess 属性的值；另一方面 Android 在 ContentResolver 中提供了 notifyChange()接口，在数据改变时会通知其他使用数据库的应用程序。

至于如何通过 ContentProvider 的方式使应用程序的数据公开，可通过两种方法，创建一个属于应用程序自己的 ContentProvider 或者将数据添加到一个已经存在的 ContentProvider 中，前提是有相同数据类型并且有写入 ContentProvider 的权限，具体的方法将在 5.3.4 节进行详细介绍。

5.3.2　通过 Content Provider 查询数据

本节来说明如何借助 Content Provider 对感兴趣的数据进行查询，前面已经介绍过，所有的 content provider 都会实现数据的查询、添加、更新和删除接口，而对于需要使用这些接口的"客户端"应用程序通常是通过 ContentResolver 对象。在应用程序中可以通过如下代码来获取一个 ContentResolver 对象实例：

```
ContentResolver cr = getContentResolver();
```

借助 ContentResolver 对象的方法，就可以方便地与感兴趣的 ContentProvider 进行交互从而获取需要的数据并进行操作。要对一个 content provider 进行确定条件的查询，需要 3 个参数作为条件：

- 用于指定 content provider 的 URI。
- 想要查询的数据项的名称。
- 需要查询的数据项对应的类型。

另外，如果仅仅是想查询数据库内的某一条记录，则还需要提供该记录的 ID。

Android 提供了两种方法用于对 content provider 进行查询，即 ContentResolver. Query()方法和 Activity. managedQuery()方法。这两种方法所需要的参数是一致的并且它们的返回值也都是一个 Cursor 对象。两者的不同之处是：managedQuery()方法所返回的 Cursor 对象的生命周期是托管给 Activity 控制，而 Query()方法所返回的 Cursor 对象的生命周期则是由程序员自己控制的。由 Activity 所管理的 Cursor 能够自动处理大量的细节，例如在 Activity 处于暂停状态时自动卸载，在 Activity 重新开始运行时重新执行查询等。对于没有被 Activity 自动管理的 Cursor 可以使用 Activity. startManagingCursor()方法将其托管给 Activity。

```
public final Cursor query (Uri uri, String[ ] projection, String selection, String[ ]
selectionArgs, String sortOrder)
public final Cursor managedQuery (Uri uri, String[ ]projection, String selection, String[ ]
selectionArgs, String sortOrder)
```

两个方法的原型如上，它们的第一次参数是用于确定 provider 的 URI，例如 Android 提供了一些自带的 provider 的 URI 常量：

```
android.provider.Contacts.Phones.CONTENT_URI——指向电话号码
android.provider.Contacts.Photos.CONTENT_URI——指向照片
```

如 5.2 节对 URI 进行说明时所示，如果仅仅需要查询某一条记录，则可以在 URI 后面附加上该条记录的 ID 信息，例如，如果要查询的记录 ID 为 1，那么 URI 的形式就类似于：

```
content://….../1
```

当然，上面提到的这种方法比较死板，不方便在常量后面添加 id，并且在各种方式中相对容易出错，因此 Android 还提供了用于添加 id 的辅助方法——ContentUris. withAppendedId()

和 Uri.withAppendedPath()，使用这两种方法就可以很方便地在 URI 后面添加 ID 数据。使用方法如下：

```
Uri person23 = ContentUris.withAppendedId(People.CONTENT_URI, 23);
Uri person23 = Uri.withAppendedPath(People.CONTENT_URI, "23");
```

下面接着来看 query()方法和 managedQuery()方法的后面几个参数，这些参数就是很详细的查询条件，如下：

- String[]projection——需要返回的数据列名称，如果该参数为 null 则会返回所有列。例如在查询联系人时可以将_ID、NUMBER、NAME 等列名作为参数。
- String selection——相当于 SQL 语句中的 WHERE 条件语句(不包括 WHERE 关键字本身)，null 值则返回所有条目。
- String[]selectionArgs——selection 的参数，依次替代 selection 字符串中的“?”。
- String sortOrder——指定查询结果的排列顺序。

用法示例：

```
Uri uri = ContactsContract.Contacts.CONTENT_URI;
String[] projection = new String[] {
ContactsContract.Contacts._ID,
ContactsContract.Contacts.DISPLAY_NAME
};
String selection = ContactsContract.Contacts.IN_VISIBLE_GROUP + " = '" +
                  "1" + "'";
String[] selectionArgs = null;
String sortOrder = ContactsContract.Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC";
return managedQuery(uri, projection, selection, selectionArgs, sortOrder);
```

上面的代码实现的功能是查询联系人 provider 并获得一个包含了 IN_VISIBLE_GROUP 值为 1 的所有联系人的 ID 和姓名的 Cursor。使用的方法是 managedQuery()。代码来自于 Contacts_VCard 示例下的 ChooseContactsForExport 类的 getContacts 方法，该方法的功能即是返回一个联系人姓名列表，效果如图 5-14 所示。

那么，如何从 managedQuery()方法所返回的 Cursor 对象中读取出查询结果数据呢？这就需要使用最基本的读取 Cursor 数据的方法，参考代码为 ChooseContactsForExport 类的 populateContactList 方法，具体代码如下：

```
Cursor cursor = getContacts();
String[] fields = new String[] { ContactsContract.Data.DISPLAY_NAME };
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
android.R.layout.simple_list_item_multiple_choice, cursor, fields
     , new int[] {android.R.id.text1});
setListAdapter(adapter);
final ListView listView = getListView();
listView.setItemsCanFocus(false);
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);     //设置多选模式
listView.setTextFilterEnabled(true);      //在 ListView 上输入字母,就会自动筛选出以此内容
                                          //开头的 Item
```

5.3.3 通过 Content Provider 修改数据

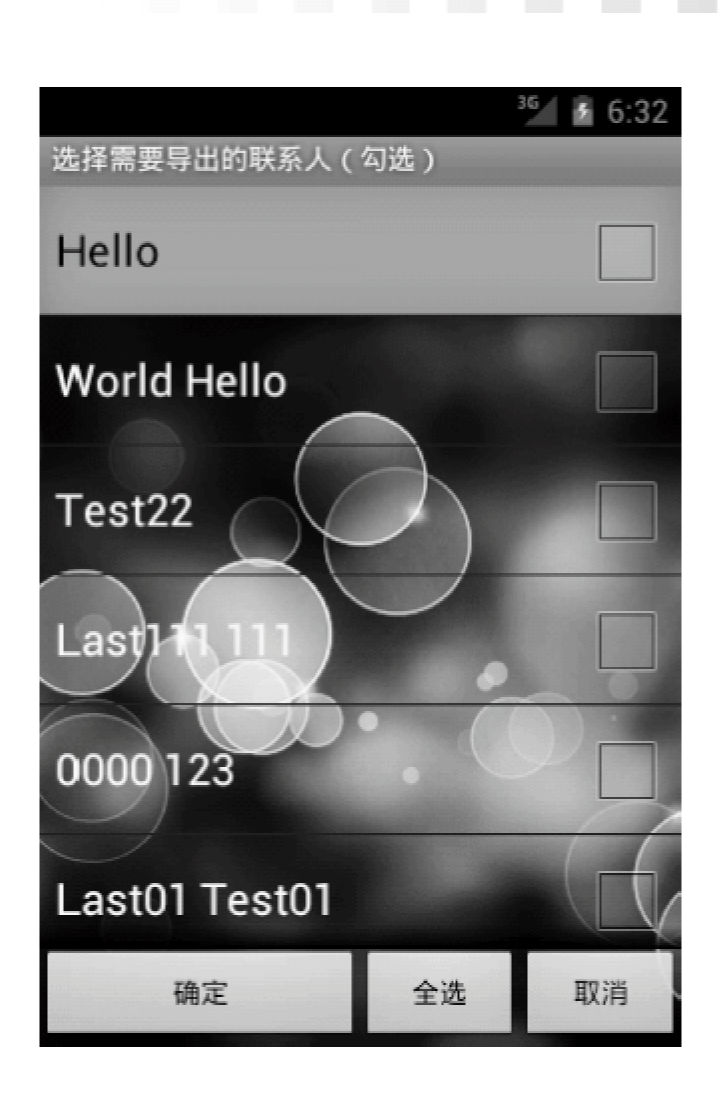


图 5-14 查询联系人 provider

通过 content provider 所提供的数据可以执行如下一些操作：

（1）向数据库中添加新的条目。

（2）向已存在条目中添加新的值。

（3）批量修改已存在的条目。

（4）删除条目。

这些操作都可以通过 ContentResolver 的方法来完成，前提是需要有对应数据的响应权限(写数据)。对应的方法如下：

- ContentResolver.insert()——第(1)条和第(2)条操作。
- ContentResolver.update()——第(3)条操作。
- ContentResolver.delete()——第(4)条操作。

1. 添加新条目

要向 content provider 中添加一条新条目，首先需要一个包含有待添加数据的 key-value 键值对数据的 ContentValues 对象，这个对象中的 key 值就对应了数据库的属性名(列名)，value 值就对应了数据库对应属性的实际值。然后通过调用 ContentResolver.insert()方法来向指定的 URI 中插入这条新的 ContentValues 键值对。insert()方法的原型如下：

```
public final Uri insert (Uri url, ContentValues values)
```

参数列表：

- url——待插入条目的表所对应的 URL。
- values——新插入条目的键值对，如果该值为空，将插入一个空条目。

返回值：

- 返回新创建条目所对应的 URL，即该表的 URL 再加上 ID 信息。通过该返回值 URL 可以得到一个指向该条目的 Cursor，之后可以通过 Cursor 访问该条目。

代码示例如下：

```
import android.provider.Contacts.People;
import android.content.ContentResolver;
import android.content.ContentValues;
ContentValues values = new ContentValues();
// 添加一个姓名为"示例"的条目并添加到 favorites
values.put(People.NAME, "示例");
// 1 = 将新条目添加至 favorites
// 0 = 不将新条目添加至 favorites
values.put(People.STARRED, 1);
Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
```

2. 向条目中添加新的值

为了丰富条目的信息，有时候需要向指定的已存在的条目添加新的数据或者是修改已有的数据。例如，向一个联系人条目中增加一条住宅电话、一个工作用 E-mail 地址等。需要进行如上操作时，使用的也是 ContentResolver.insert()方法，由于仅仅是添加一项属性，因此需要更细化 url 参数，使操作可以定位到一条属性，对于 Contacts，Android 提供一个常量 CONTENT_DIRECTORY 来表示这一段需要附加的 url，使用方法如下。

接第 1 部分的示例，需要对上一步已经新建的条目添加 Phone 和 E-mail 信息，可以通过如下步骤：

```
Uri phoneUri = null;
Uri emailUri = null;
// 修正前面返回的 uri,得到插入 phone 的 uri
phoneUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);
values.clear();
values.put(People.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);
// 用类似的方法添加 email
emailUri = Uri.withAppendedPath(uri, People.ContactMethods.CONTENT_DIRECTORY);
values.clear();
// ContactMethods.KIND 用于指定添加数据的类型,因为 E-mail 不是一个主要的类型
values.put(People.ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(People.ContactMethods.DATA, "test@example.com");
values.put(People.ContactMethods.TYPE, People.ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);
```

3. 更新和删除条目

更新条目的方法是 ContentResolver.update()，该方法的原型是：

```
public final int update (Uri uri, ContentValues values, String where, String[] selectionArgs)
```

参数列表：

- uri——需要进行更新的数据对应的 uri。
- values——用于更新的结果数据，如果为空，则删除当前值。
- where——查询条件。
- selectionArgs——查询条件语句的参数。

返回值：

- 被更新的条目数。

删除条目的方法是 ContentResolver.delete()，该方法的原型为：

```
public final int delete (Uri url, String where, String[] selectionArgs)
```

参数列表：

- url——欲删除条目所对应的 url。

- where——删除符合条件的一系列条目时所需要的查询条件。

返回值：

- 删除的条目数。

这两个方法的具体使用与前面两种方法类似，需要注意的就是在批量更新或者批量删除的操作时需要配合合适的 where 条件语句，避免由于该条件语句的错误而导致删除或改变并不打算删除或改变的数据。

5.3.4 创建 Content Provider

本节将介绍如何创建一个 Content Provider，在 Android SDK 提供的 samples 中的 NotePad 示例中就实现了一个 ContentProvider，因此此处不再提供示例，建议读者结合 NotePad 示例中 NotePadProvider.java 的代码来了解本节内容。

要创建一个可供使用的 content provider，需要完成如下工作：

(1) 正确配置一个用于存储数据的机构。通常可以使用文件存储的方式或者是 SQLite 数据库的方式，当然也可以使用其他任何可用的数据存储。这里推荐使用的是 SQLite 数据库，具体用法如前所述。

(2) 继承 ContentProvider 类实现一个子类用于提供对数据访问的接口。

(3) 在 Project 的 AndroidManifest.xml 文件中声明所实现的 content provider。

1. 继承 ContentProvider 类

通过继承 ContentProvider 类来实现子类可以将应用程序私有的数据暴露成外部接口供其他对象调用(ContentResolver 对象或 Cursor 对象)。一般地，继承 ContentProvider 类需要实现如下几个抽象方法：

- query()——查询数据。
- insert()——插入数据。
- update()——更新数据。
- delete()——删除数据。
- getType()——通过 uri 得到数据的类型。
- onCreate()——程序入口。

其中，query()方法需要返回一个能够通过迭代遍历查询结果数据的 Cursor。Cursor 实际上仅仅是一个接口类型，但是 Android 已经事先定义好了许多可直接使用的 Cursor，例如用于遍历 SQLite 数据库数据的 SQLiteCursor，可以通过调用 SQLiteDatabase 类的 query()方法来获取这类的 Cursor 对象。

由于 ContentProvider 类的方法是暴露出来给应用程序调用的，因此它可能被多个进程或线程同时调用，所以这些方法的实现必须是线程安全的，另外，为了保证使用数据的客户程序对数据的更改的知情权，最好在对数据进行修改之后调用 ContentResolver.notifyChange()方法来通知所有监听数据更改的监听器。

除了需要实现如上所述的一些方法之外，还需要注意如下几点，以方便客户程序使用该 provider。

(1) 定义一个名为 CONTENT_URI 的 public static final Uri 类型的常量，该常量用于

代表所实现的 content provider 类所处理的数据地址，其他应用程序通过这个 Uri 来指定对该部分数据的请求访问。因此该常量在整个系统中必须是唯一的，确保这个常量唯一的较好方法是使用该 content provider 类完整的名称（包含完整包名和类名）。例如，SDK Samples 中 NotePad 示例的 CONTENT_URI 被定义为：

```
public static final Uri CONTENT_URI = Uri.parse(SCHEME + AUTHORITY + PATH_NOTES);
```

其中：

```
private static final String SCHEME = "content://";
public static final String AUTHORITY = "com.google.provider.NotePad";
private static final String PATH_NOTES = "/notes";
```

它额外还增加了字符串/notes，这相当于对应用程序内的数据进行了进一步的分类，方便后面其他类型数据的扩展。

(2) 定义数据的列名，与数据库的定义一样，这些列名即代表了数据的属性，这些列名在客户应用程序对数据进行查询时由 content provider 返回。如果应用程序使用数据库如 SQLite 来存取数据，那么列名就直接表现为 SQLite 数据库中相应的列名。如果需要自定义列名，则也需要使用 public static 类型的字符串常量来指定数据的列。

需要注意的是，必须包含一个整型并且名为“_id”的列（用字符串常量_ID 代表），这个列是用于唯一的数据库中每一条记录的 id，无论是否存在其他的数值唯一的列，都必须包含这个列。如果使用的是 SQLite 数据库，那么这个_ID 字段的类型应该被定义为：

```
INTEGER PRIMARY KEY AUTOINCREMENT
```

(3) 对每一列的数据类型进行详尽的文档注释。

(4) 如果创建的 content provider 是用于处理非通用的数据类型，那么就必须定义一个新的 MIME 类型来代表该数据并供 getType()方法返回。Android 平台将根据提交给 getType()方法的 Uri 所返回的 MIME 类型来决定将这个请求指向一个单独的记录或者是一组记录。因为单个记录的 MIME 类型和多个记录的 MIME 类型是不同的，它们的规则如下，同样取自于 NotePad 示例。

• 多个记录的 MIME type：

```
"vnd.android.cursor.dir/vnd.google.note"
```

• 单个记录的 MIME type：

```
"vnd.android.cursor.item/vnd.google.note"
```

两者的区别就在第一部分，“/”前面的部分是系统识别的部分，就相当于定义一个变量时的变量数据类型，通过这个“数据类型”，系统能够知道所要表示的单个项还是一组项。而“/” 后面的部分就是用来指定特定的数据了。

在 getType()方法被调用时，如果参数 Uri 是：

```
content://com.google.provider.NotePad/notes/1
```

则会返回 MIME type：

```
vnd.android.cursor.item/vnd.google.note
```

若 Uri 是：

```
content://com.google.provider.NotePad/notes
```

则会返回 MIME type：

```
vnd.android.cursor.dir/vnd.google.note
```

这里所需的 Uri 的解析和匹配工作借助于 UriMatcher 类来完成，这里通过一些代码片段来简要说明如何使用 UriMatcher 类，如下的代码片段截取自 Android Reference 中的 android.content.UriMatcher 类的说明文档：

```
01  private static final int PEOPLE = 1;
02  private static final int PEOPLE_ID = 2;
03  private static final int PEOPLE_PHONES = 3;
04  private static final int PEOPLE_PHONES_ID = 4;
05  private static final UriMatcher sURIMatcher = new UriMatcher(UriMatcher.NO_MATCH);
06  static
07  {
08      sURIMatcher.addURI("contacts", "people", PEOPLE);
09      sURIMatcher.addURI("contacts", "people/#", PEOPLE_ID);
10      sURIMatcher.addURI("contacts", "people/#/phones", PEOPLE_PHONES);
11      sURIMatcher.addURI("contacts", "people/#/phones/#", PEOPLE_PHONES_ID);
12  }
13  public String getType(Uri url)
14  {
15      int match = sURIMatcher.match(url);
16      switch (match)
17      {
18          case PEOPLE:
19              return "vnd.android.cursor.dir/person";
20          case PEOPLE_ID:
21              return "vnd.android.cursor.item/person";
22          case PEOPLE_PHONES:
23              return "vnd.android.cursor.dir/phone";
24          case PEOPLE_PHONES_ID:
25              return "vnd.android.cursor.item/phone";
26          default:
27              return null;
28      }
29  }
```

如上面的代码所示，首先在第 01～04 行定义了 4 个具名常量，它们作为 4 种不同的 Uri 匹配类型的编码使用，第 05～12 行代码则是新建了一个 UriMatcher 对象并通过它的

addURI()方法定义了 4 种匹配类型，addURI 方法的原型为：

```
void addURI(String authority, String path, int code)
```

这个方法将一个 URI 添加至 UriMatcher 的匹配列表中，添加成功后再对 URI 进行匹配时，将会返回对应的 code 码。

参数列表：

- authority——授权字符串，是用于授权该 provider 的字符串，如此处的 contacts。
- path——用于匹配 Uri 的路径名，在这个字符串中可以使用星号“*”代表任意的字符串，使用井号“#”代表任意的数字。
- code——这个参数用于设置当 path 匹配成功时将返回的编码。

使用 addURI()方法建立了匹配规则之后，就可以实现 getType()方法了，如上面代码的第 15～27 行所示，首先通过 UriMatcher 的 match 方法获取匹配 Uri 后返回的编码，然后再根据该编码返回对应的 MIME 类型字符串。match 方法的原型为：

```
int match (Uri uri)
```

参数列表：

- uri——待匹配的 uri。

返回值：

- int——匹配结果，即前面设定的编码。

(5)如果需要由 provider 提供的数据是类似于大体积的图片文件的类型，这类数据由于过大而不方便直接存放在数据库的表中，这时候往往在这个字段上存放一个 URI 字符串，即在数据库中仅存放可定位该资源的 URI 信息。在这种情况下还需要在该条记录中增加一个“_data”字段用于存放该文件实际的存放路径，这个字段仅供 ContentResolver 调用，而客户程序则通过调用 ContentResolver 的 openInputStream()方法来访问该数据，对客户程序来说就仿佛文件直接存放在数据库表中一样。

2. 声明 Content Provider

既然 content provider 是把数据暴露给外界使用的方式，那么就必须要使这个 provider 对于外界可见，如何让外界知道这个 content provider 呢？就需要通过在 AndroidManifest.xml 文件中进行声明的方式把该 provider 告知给 Android 系统，从而由系统来维护 provider 的可见性。声明的方式是在相应的应用程序的 AndroidManifest.xml 中增加一个<provider>标签，如果一个 provider 没有在 AndroidManifest.xml 中进行声明，那么对于外界来说它就是不存在的，类似于 Activity 的声明，如果一个 Activity 没有在 AndroidManifest.xml 文件中进行声明，那么这个 Activity 就不会被系统所识别。

<provider>标签包含了如下两个主要属性：

- name——这个属性值应该指定为对应 provider 类的全名。
- authorities——如前面已经提到过，这个属性是属于 CONTENT_URI 的一部分，用于代表该 provider 的唯一识别字符串。

一个 provider 的声明示例如下：

```
<provider android:name = "NotePadProvider"
android:authorities = "com.google.provider.NotePad"
……
</provider>
```

<provider>还包括了其他的一些属性，例如设置读写数据的权限、设置一个代表该 provider 的图标等，具体信息参考 SDK 文档的 Dev Guide 栏下 Framework Topic 分类下的 The AndroidManifest.xml File 小节。

3. 创建数据存储

由于 content provider 的数据存取使用的是数据库的语义，因此大部分情况下都是使用 SQLite 数据库作为 provider 的数据存储，本节简要介绍一个初始化数据存储(SQLite)的示例。

通常在 ContentProvider 类的 onCreate()方法中对数据库进行创建，这个方法将在 content provider 初始化时被调用。在 NotePad 示例中，onCreate()方法将创建一个到数据库的连接，如果指定数据库不存在则新建数据库。代码如下：

```
@Override
public boolean onCreate() {
     mOpenHelper = new DatabaseHelper(getContext());
     return true;
}
static class DatabaseHelper extends SQLiteOpenHelper {
  DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }
  @Override
  public void onCreate(SQLiteDatabase db) {
       db.execSQL("CREATE TABLE " + NotePad.Notes.TABLE_NAME + " ("
            + NotePad.Notes._ID + " INTEGER PRIMARY KEY,"
            + NotePad.Notes.COLUMN_NAME_TITLE + " TEXT,"
            + NotePad.Notes.COLUMN_NAME_NOTE + " TEXT,"
            + NotePad.Notes.COLUMN_NAME_CREATE_DATE + " INTEGER,"
            + NotePad.Notes.COLUMN_NAME_MODIFICATION_DATE + " INTEGER"
            + ");");
  }
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            db.execSQL("DROP TABLE IF EXISTS notes");
            onCreate(db);
      }
  }
```

代码通过一个 DatabaseHelper 类来帮助连接数据库，该类继承自 SQLiteOpenHelper，可以看到在 ContentProvider 的 onCreate()方法中只需要初始化一个 DatabaseHelper 类型的 mOpenHelper 对象即可，在程序的其他部分需要使用数据库时，直接借助该对象就行了。

对于前面所提到的大文件的存储方式，这里也做一个简单的示例，如果把大文件(图片、音视频等)直接存放到数据库中，则可能是如下的形式：

```
CREATE TABLE user (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    password TEXT,
    picture BLOB
);
```

这种方式在原理上是可行的，但是对于大文件的读取，通过文件系统直接读取的速度要远远大于从数据库中读取，但是由于 Android 限定一个应用程序不能够访问由其他应用程序所创建的文件，因此需要采用如下的一种“迂回”的手段，这时需要增加另一个表，类似如下代码：

```
CREATE TABLE user (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    password TEXT,
    picture TEXT
);

CREATE TABLE userPicture (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    _data TEXT
);
```

如上述代码所示，在原先的表中的 picture 字段现在存放的是一个指向 userPicture 表中某个条目的 Uri 字符串，而在 userPicture 表的_data 字段存放的则是实际的文件路径；通过这样的方式可以使得打开文件的操作是由 ContentProvider 执行而非由客户程序执行，从而避免了出现权限不足的问题，如果把文件路径直接存放到 user 表中，则客户程序会得到对应文件的存放路径但是却没有权限打开。

5.3.5 使用 NotePadProvider

经过 3.3.4 节中介绍的一系列步骤之后，NotePad 应用程序就可以将其内部数据通过 NotePadProvider 供外部应用程序访问了，为了验证这一点，本节将实现另一个示例应用，用于测试 NotePadProvider 的可用性。

首先需要修改的是 NotePadProvider 的访问权限，打开 NotePad 项目的 AndroidManifest 文件，可以发现 NotePadProvider 的声明方式为：

```
<provider android:name = "NotePadProvider"
    android:authorities = "com.google.provider.NotePad"
    android:exported = "false">
    <grant - uri - permission android:pathPattern = ". * " />
</provider>
```

其中包含的标签和属性的含义如下：

- android:name——实现该 provider 的类名。
- android:authorities——该 provider 的授权字符串，对应前面提到的 URI 的 B 部分。
- android:exported——该 provider 的可见性，如果为 false 则只能被本应用自身使用，若为 true 则能够被外部应用使用，如果不明确指定该属性，它的默认值为 true，即默认允许外部应用程序使用。
- <grant-uri-permisson>——该标签是配合属于 provider 的 android:grantUriPermissions 属性使用的，android:grantUriPermissions 的取值表明的是该 content provider 的访问权是否可以被赋予那些本不具有访问权的对象，这个权限的赋予操作可以临时忽视掉 readPermission、writePermission、permission 这 3 个属性的限制，如果它的取值为 true，则表示允许授予 content provider 所有内容访问权限；如果取值为 false，则表示仅允许授予 content provider 部分内容的访问权限，而这个可以被授予访问权限的部分就由<grant-uri-permisson>标签来决定。android:grantUriPermissions 的默认取值为 false。此处不需要对 provider 的访问权限进行过于严格的限制，因此可以忽略掉该标签的内容。

为了使外部应用程序能够访问该 provider，修改 android:exported 的值为 true 即可：

```
android:exported = "true"
```

在修改了 android:exported 的值之后，打开 NotePad 应用，在其中添加并保存几个文本，目的是向 NotePad 的 SQLite 数据库插入一些数据，以备测试使用。单击界面右上角的新建笔记按钮，新建并保存若干笔记(如图 5-15 所示)，默认的笔记标题取的是内容的前几个字符，可以在主界面通过长时间按某个笔记的方式弹出上下文菜单并选择修改笔记的标题(如图 5-16 所示)。新建几个笔记之后可以得到类似如图 5-17 所示的列表。

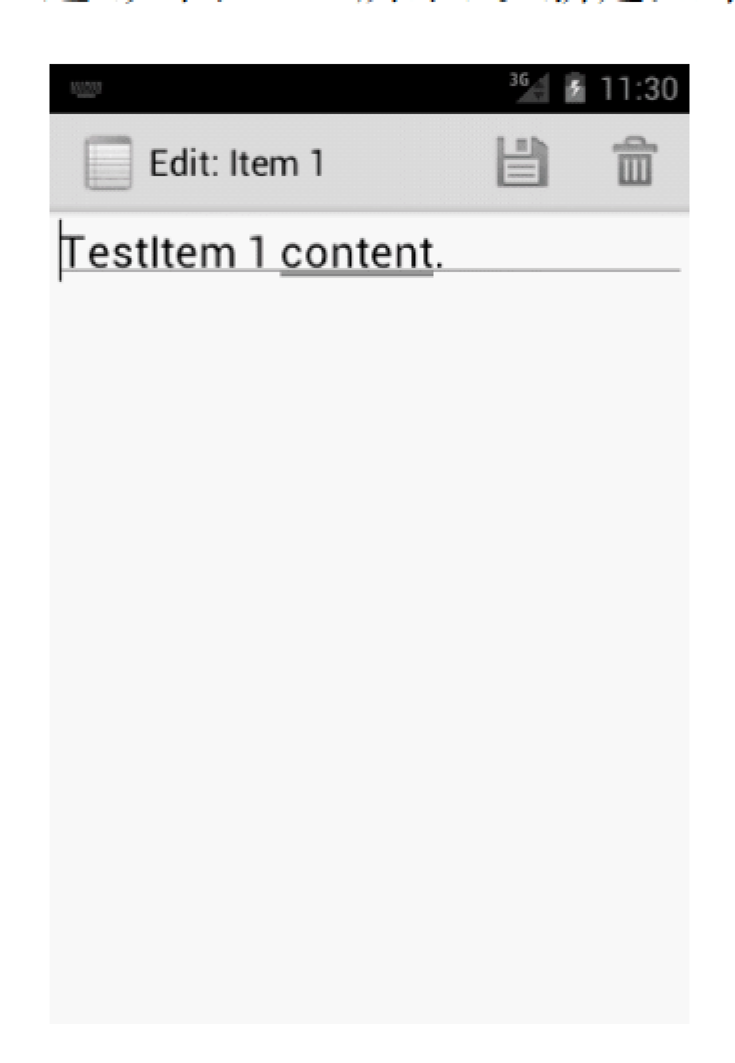


图 5-15　新建笔记

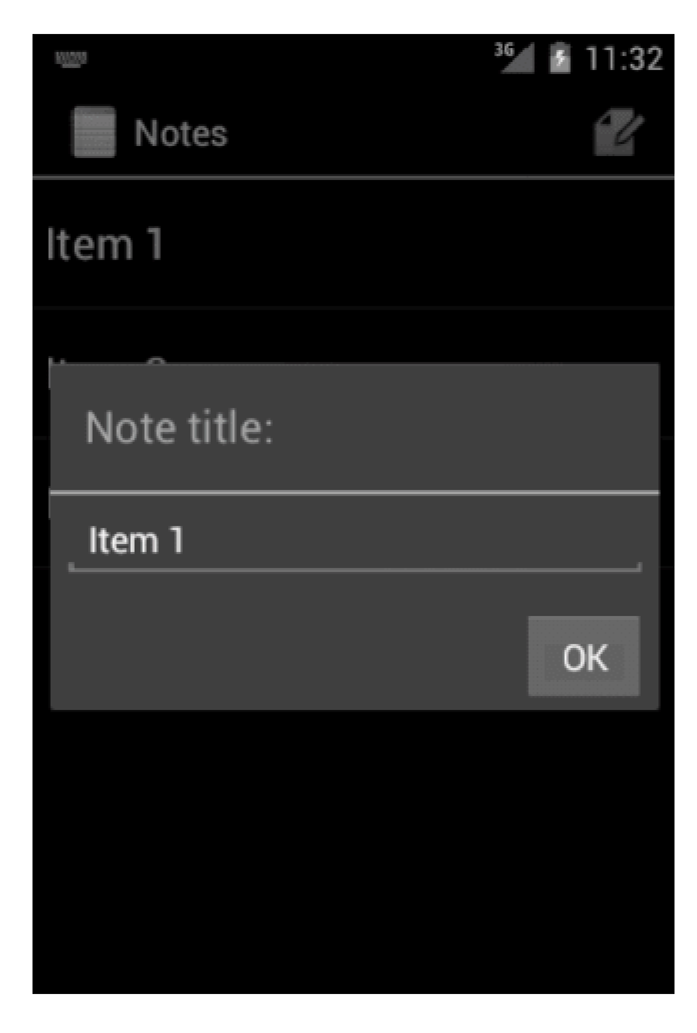


图 5-16　修改笔记标题

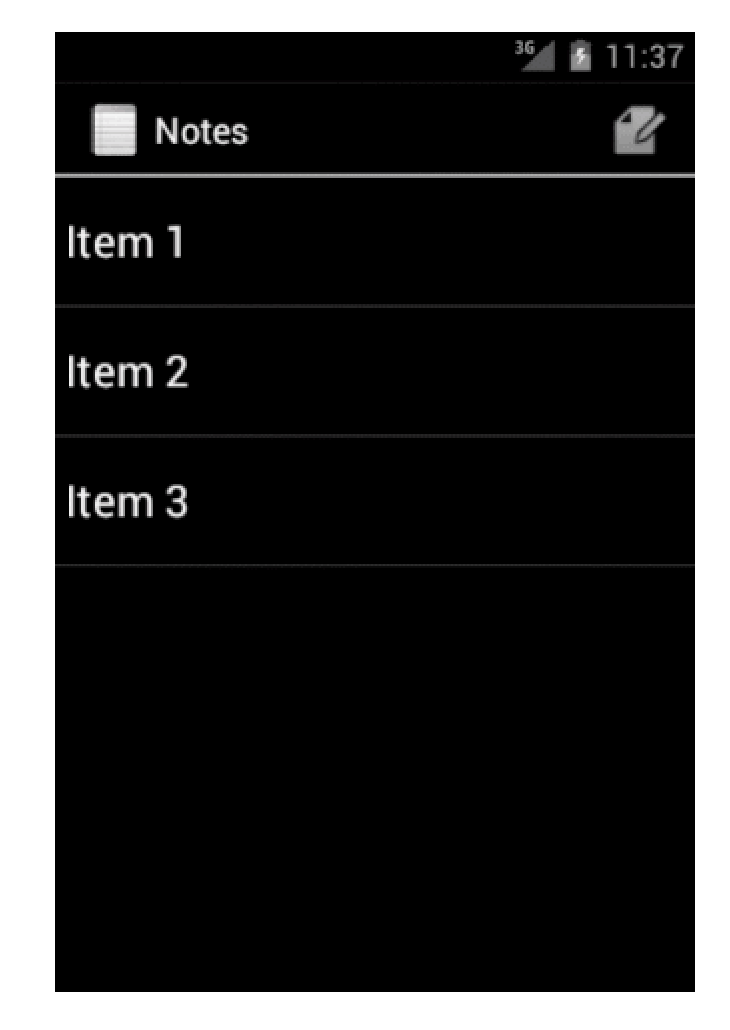


图 5-17　笔记列表

现在的 NotePad 数据库中就已经存在 3 条记录了，要测试外部应用程序是否可以访问 NotePadProvider，在 Eclipse 中新建一个空白项目，然后在该项目的 Activity 中添加如下测试用方法：

```
public void testNotePadProvider() throws Throwable {
        ContentResolver contentResolver = this.getContentResolver();
        Uri uri = Uri.parse("content://com.google.provider.NotePad/notes");
        Cursor cursor = contentResolver.query(uri, new String[] { "_id",
              "title", "note" }, null, null, "_id desc");
        while (cursor.moveToNext()) {
            Log.i("testnotepad", "_id = " + cursor.getInt(0) + ",title = "
                  + cursor.getString(1) + ",note = " + cursor.getString(2));
        }
}
```

该方法的作用是查询 NotePadProvider 并使其返回所有条目的_id、title 及 note 字段内容，然后使用 Log 函数输出。

在新建项目的 Activity.onCreate()方法加入对上述方法的引用：

```
try {
   testNotePadProvider();
} catch (Throwable e) {
   e.printStackTrace();
}
```

然后运行项目，在 Eclipse 的 LogCat 视图中将会发现类似如图 5-18 所示的信息。

Level	PID	Application	Tag	Text
I	693	com.se...	testnotepad	_id=3,title=Item 3,note=Content 3.
I	693	com.se...	testnotepad	_id=2,title=Item 2,note=Content 2.
I	693	com.se...	testnotepad	_id=1,title=Item 1,note=TestItem 1 content.

图 5-18 访问 NotePadProvider 成功

可以发现，通过新建项目成功访问了 NotePadProvider 并获取了返回值，从而验证了 ContentProvider 的实现。

参考文献

1. 开放源码嵌入式数据库 SQLite 简介：http://www.ibm.com/developerworks/cn/opensource/os-sqlite/.
2. Rick Rogers, John Lombardo. Android Application Development. 1st Edition. O'Reilly Media, Inc. 2009-05-26.

第6章 多进程与多线程

6.1 进程与线程概念

6.1.1 什么是进程

进程(Process)是计算机中已运行程序的实体。程序本身只是指令的集合(静态),进程才是真正被执行的指令(动态)。若干进程有可能与同一个程序有联系,并且每个进程都可以按同步(顺序)或不同步(平行)的方式独立运行。现代计算机系统可在同一时间内加载多个程序和进程到内存中,凭借时间共享(或称多任务)的方式,在单一处理器上表现出同时(并行性)运行的特性。另外,对于采用多线程技术的操作系统或计算机架构,上述的平行进程可在多 CPU 主机上实现真正同时运行。

多道程序在执行时,需要共享系统资源,从而导致各程序在执行过程中出现相互制约的关系,程序的执行表现出间断性的特征。这些特征都是在程序的执行过程中产生的,是一个动态的过程,而传统的程序本身只是一组指令的集合,是一个静态的概念,因此无法描述程序在内存中的执行情况,仅从程序上不能看出它何时执行、何时停顿,也无法看出它与其他执行程序的关系,因此,程序这个静态概念已不能如实反映程序并发执行过程的特征。为了深刻描述程序动态执行过程的性质,引入了“进程”的概念。

简单地说,进程可以被描述为程序关于某个数据集合的一次运行活动,它使用程序计数器、寄存器来记录当前活动的信息,它是分时系统的基本运作单位,同时也是资源管理及分配的最小单元。进程是“执行中的程序”,是活动的,而程序是一个没有生命的实体,只有处理器赋予程序生命时,它才能成为一个活动的实体,我们称其为进程。

进程是一种实体(可理解为一种数据结构),它包括:

- 文本区域(text region)——存储处理器执行的代码。
- 数据区域(data region)——存储变量和进程执行期间使用的动态分配的内存。
- 堆栈(stack region)——存储活动所调用的指令和本地变量。

6.1.2 进程的特征

进程具有如下几点特征:

- 动态性——进程的实质是程序的一次执行过程,进程是动态产生、动态消亡的。
- 并发性——任何进程都可以同其他进程一起并发执行。

- 独立性——进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。
- 异步性——由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。

6.1.3 进程的状态及状态切换

进程在运行时，它的状态（state）会发生改变。一个进程能够处于如下几种状态：

- 新建（new）——进程新产生中。
- 运行（running）——正在运行。
- 等待（waiting）——等待某事发生，例如等待用户输入完成。
- 就绪（ready）——等待 CPU 调度。
- 退出（terminated）——运行结束。

进程在这几种状态间的切换方式如图 6-1 所示。

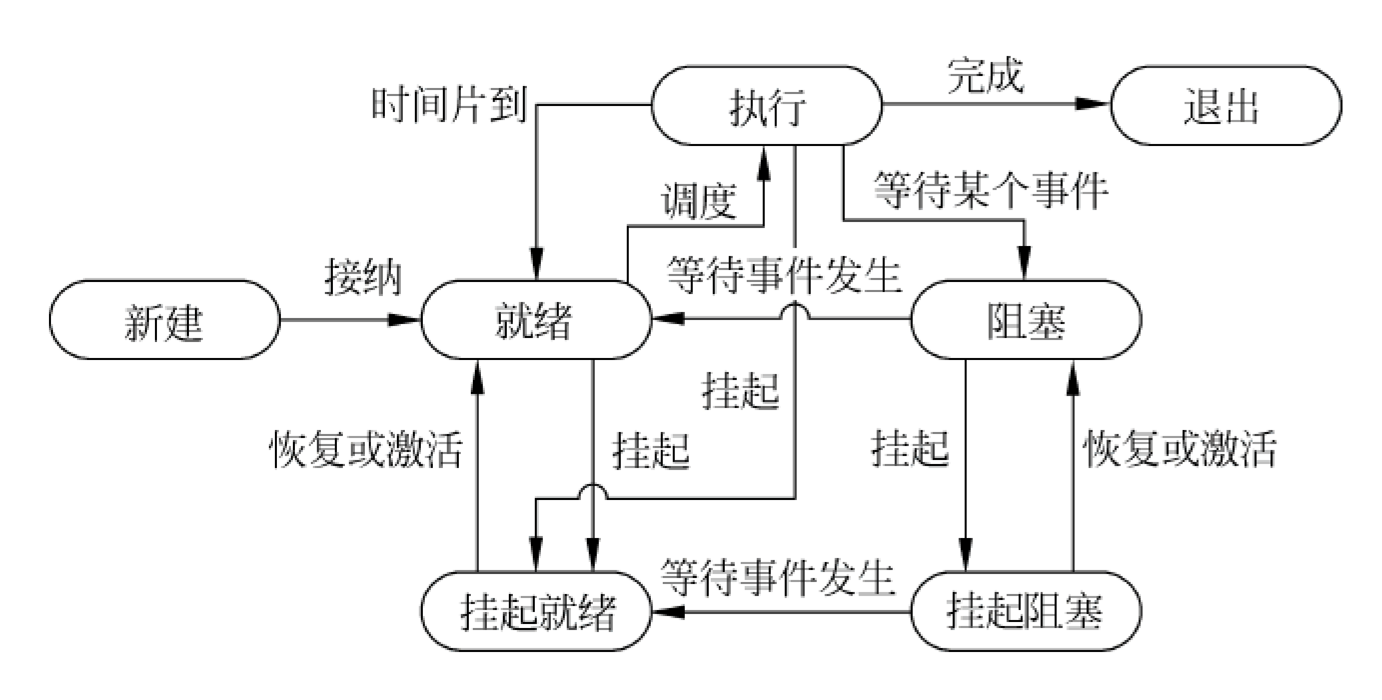


图 6-1 进程的状态转换关系

在内存资源紧张的情况下，如何处理多个进程竞争内存资源的问题呢？通常有如下两种解决方法：

- 采用交换技术——换出一部分进程到外存，来腾出内存空间。
- 采用虚拟存储技术——每个进程只能装入一部分程序和数据（存储管理部分）。

交换技术（Swapping）是指：将内存中暂时不能运行的进程，或暂时不用的数据和程序，换出到外存，以腾出足够的内存空间，把已具备运行条件的进程，或进程所需要的数据和程序，换入内存。通常不允许将进程的 PCB（进程控制块）换到外存去，因为 PCB 是进程存在与否的唯一的标志，因此交换技术是把进程所需的数据和程序换出到外存。

这里对进程的挂起状态进行一下说明，当进程被交换到外存后，它的状态变为挂起状态。被挂起的进程不能立即执行，直到它等待的某个事件发生，只有挂起它的进程才能使之由挂起状态转换为其他状态。

6.1.4 什么是线程

线程（Thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中某个单一顺序的控制流。线程必须有一个父进程，而一个进程中可以包含若干个线程，

线程与父进程的其他子线程共享该进程所拥有的全部资源，这些子线程共享相同的内存地址空间，可以访问相同的变量和对象。线程由父进程创建和撤销，从而实现程序的并发执行。线程又被称为轻量级进程(Lightweight Process，LWP)，与进程类似，线程也具有新建、就绪、阻塞、执行和结束几种状态。

6.1.5　线程的状态及状态切换

线程的状态及状态之间的转换关系如图 6-2 所示。

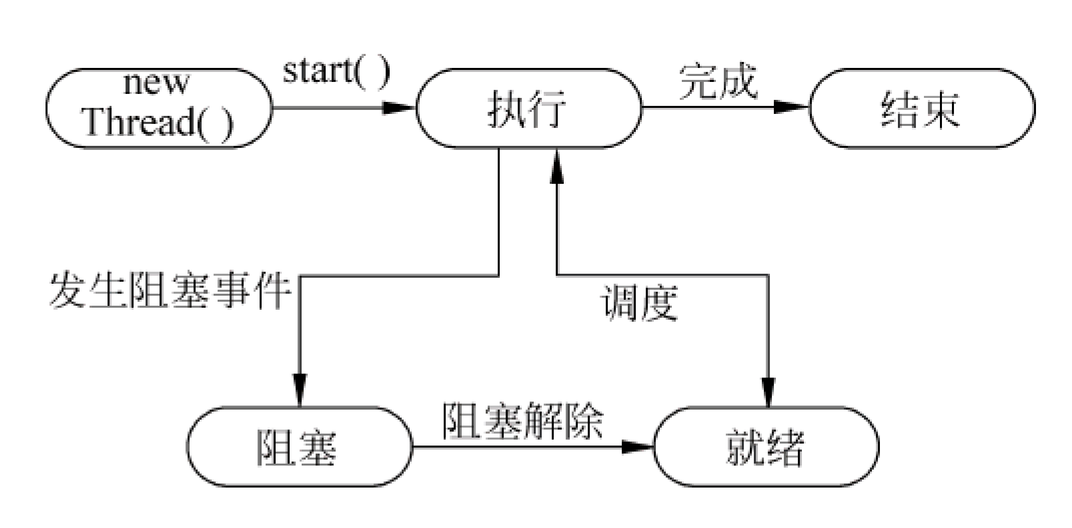


图 6-2　线程的状态转换关系

6.1.6　进程与线程的关系

进程与线程有着十分相似的特征，但是它们之间又有着区别，本节简单地介绍一下两者之间的联系与区别。

从几个方面简单地进行概括：一个程序至少有一个进程，一个进程至少有一个线程；线程的划分尺度小于进程，多线程程序的并发性较高；每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。线程不能够独立执行，必须依存在应用程序中，由应用程序提供对多个线程的控制和管理。

操作系统以进程为单位进行调度和管理以及资源分配，进程和线程的主要差别就在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其他进程产生影响；而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮；但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

进程和线程之间的关系可以使用道路和车道之间的关系来作为隐喻，有如下的描述：

- 这些线程(车道)共享了进程(道路)的公共资源(土地资源)。
- 这些线程(车道)必须依赖于进程(道路)，也就是说，线程不能脱离于进程而存在(就像离开了道路，车道也就没有意义了)。
- 这些线程(车道)之间可以并发执行(各个车道你走你的，我走我的)，也可以互相同步(某些车道在交通灯亮时禁止继续前行或转弯，必须等待其他车道的车辆通行完毕)。
- 这些线程(车道)之间谁先运行是未知的，只有在线程刚好被分配到 CPU 时间片(交通灯变化)的那一刻才能知道。
- 这些线程(车道)之间依靠代码逻辑(交通灯)来控制运行，一旦代码逻辑控制有误

(死锁,多个线程同时竞争唯一资源),那么线程将陷入混乱、无序之中。

6.1.7 多线程简介

多线程(Multithreading)技术,通常是指从软件或者硬件上实现多个线程并发执行的技术,此处主要讨论的是软件多线程。

软件多线程,是指操作系统通过快速地在不同线程之间进行切换,由于时间间隔很小,来给用户造成一种多个线程同时运行的假象。虽然现今已经有许多处理器已经从硬件上实现了对多线程的支持,但对于较上层的应用开发,主要是借助各编程语言对多线程编程的支持来实现。比较常用的语言如 C/C++、C#、Java 等都提供了对多线程的支持。

活动在不同的线程之间切换,达到多线程并发执行的效果,如图 6-3 所示。

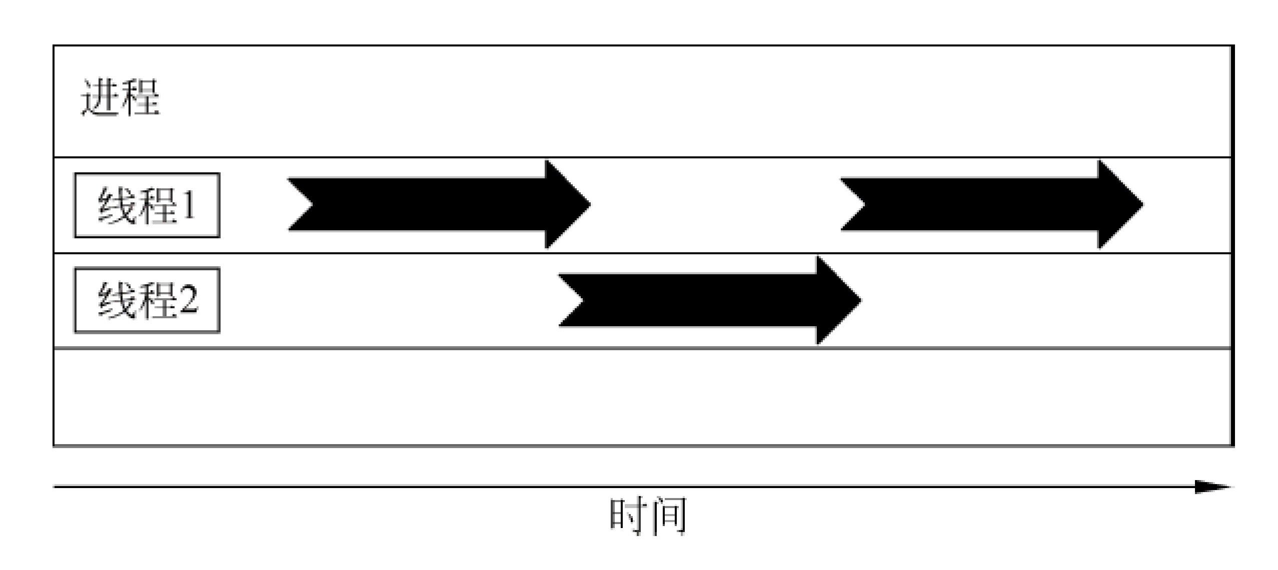


图 6-3 多线程示意

为什么要使用多线程呢?这是因为使用多线程是为了使得多个线程并行工作,从而达到同时完成多项任务,并提高系统的效率。线程就是在同一时间需要完成多项任务的时候被提出并且实现的。多线程具有如下几点好处:

- 可以把占据长时间的任务放到后台去处理。
- 使用户界面变得更加友好。
- 可能加快程序的执行速度。

6.1.8 多进程简介

与多线程一样,多进程也是一种实现并行执行任务的方法,相对于多线程优缺点比较如下:

- 进程优点——编程、调试简单,可靠性较高。
- 进程缺点——创建、销毁、切换速度慢,内存、资源占用大。
- 线程优点——创建、销毁、切换速度快,内存、资源占用小。
- 线程缺点——编程、调试复杂,可靠性较差。

由于线程可以共享父进程的所有资源,因此它们相互间的通信可以通过全局变量来实现线程通信,而由于不同进程使用各自独立的地址空间,相互不能够直接进行访问,因此就需要使用进程间通信(IPC)。IPC 有如下的几种方式:

- 管道(Pipe)及有名管道(named pipe)。
- 信号(Signal)。
- 报文(Message)队列。

- 信号量(semaphore)。
- 套接字(Socket)。
- 共享内存。

管道(Pipe)及有名管道(named pipe)：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

信号(Signal)：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux 除了支持 UNIX 早期信号语义函数 signal 外，还支持语义符合 Posix.1 标准的信号函数 sigaction(实际上，该函数是基于 BSD 的，BSD 为了实现可靠信号机制，又能够统一对外接口，用 sigaction 函数重新实现了 signal 函数)。

报文(Message)队列(消息队列)：消息队列是消息的链接表，包括 Posix 消息队列 system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

信号量(semaphore)：主要作为进程间以及同一进程不同线程之间的同步手段。

套接字(Socket)：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 UNIX 系统的 BSD 分支开发出来的，但现在一般可以移植到其他类 UNIX 系统上：Linux 和 System V 的变种都支持套接字。

共享内存：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其他通信机制，如信号量结合使用，来达到进程间的同步及互斥。

6.1.9　同步和互斥问题

在进行多进程和多线程编程时，需要注意的问题是数据同步和互斥的问题。常见的用于描述该类问题的模型有：

- 哲学家就餐问题。
- 生产者/消费者问题。
- 读者/写者问题。

在后面的章节中会对生产者/消费者模型进行详细阐述。

6.2　Android 进程与线程

6.2.1　Android 进程模型

在安装 Android 应用程序的时候，Android 会为每个程序分配一个 Linux 用户 ID，并设置相应的权限，这样就使得其他应用程序不能访问此应用程序所拥有的数据和资源。在 Linux 中，一个用户 ID 识别一个给定用户；在 Android 上，一个用户 ID 识别一个应用程

序。应用程序在安装时被分配的用户 ID,在它的存续期间内,这个用户 ID 保持不变。

在默认情况下,每个应用程序运行在它自己的 Linux 进程中。当需要执行应用程序中的代码时,Android 会启动一 DalvikVM(Java 虚拟机),即一个新的进程来执行,因此不同的 apk 运行在相互隔离的环境中。当应用程序的组件第一次运行时,Android 将启动一个只有一个执行线程的 Linux 进程。在默认情况下,应用程序所有的组件运行在这个进程和线程中。

另外,可以通过给两个应用程序分配相同的 linux 用户 ID,这样它们就能访问对方所拥有的资源了。为了保留系统资源,拥有相同用户 ID 的应用程序可以运行在同一个进程中,共享同一个 DalvikVM。要实现这个功能,首先必须使用相同的私钥签署这些应用程序,然后必须使用 manifest 文件给它们分配相同的 Linux 用户 ID,可通过用相同的值/名定义 manifest 属性 android:sharedUserId 来完成这一操作。

Android 的组件运行于哪个进程中由 AndroidManifest. xml 控制。组件元素——<activity>、<service>、<receiver>、<provider>,都有一个 process 属性可以指定组件运行在哪个进程中。这个属性可以设置为每个组件运行在自己的进程中,或者某些组件共享一个进程而其他的不共享。它们还可以设置为不同应用程序的组件运行在同一个进程中——即假设这些应用程序共享同一个 Linux 用户 ID 且被分配了同样的权限。<application>元素也有 process 属性,为所有的组件设置一个默认值。

所有的组件都在特定进程的主线程中实例化,且系统调用组件是由主线程派遣。不会为每个实例创建单独的线程,因此,对应这些调用的方法——诸如 View. onKeyDown()报告用户的行为和生命周期通知,总是运行在进程的主线程中。这意味着组件不应该在被系统调用时去执行很长一段时间或执行有阻塞的操作(如网络操作或循环计算),因为这将阻塞进程中的其他组件,造成用户界面假死的情况。因此,需要为这些费时较长的操作衍生出独立的线程来执行。

1. Android 对进程的回收

当内存剩余较小且其他进程请求较大内存并需要立即分配,Android 要回收某些进程,进程中的应用程序组件会被销毁。当它们再次运行时,会重新创建一个新进程。当决定终结哪个进程时,Android 会权衡它们对用户的重要性。例如,前台运行进程的重要性高于后台进程;与运行在屏幕可见的活动进程相比(前台进程),它更容易关闭一个活动在屏幕是不可见区域的进程(后台进程)。决定是否终结进程,取决于运行在进程中的组件状态。

2. Android 系统进程

init 进程(1 号进程),父进程为 0 号进程,执行根目录底下的 init 可执行程序,是用户空间进程,kthreadd 进程(2 号进程),父进程为 0 号进程,是内核进程,其他内核进程都是直接或者间接以它为父进程。

下面介绍一下 Android 提供的一个系统关键服务 Zygote。

Zygote 的基本翻译含义——[生物]受精卵,接合子,接合体。

从字面上就可以大体知道 Zygote 的作用了。即通过 Zygote 产出不同的子 Zygote。从大的架构上讲,Zygote 是一个简单的典型 C/S 结构。其他进程作为一个客服端向 Zygote 发出“孵化”请求,Zygote 接收到命令就“孵化”出一个 Activity 进程来,如图 6-4 所示。

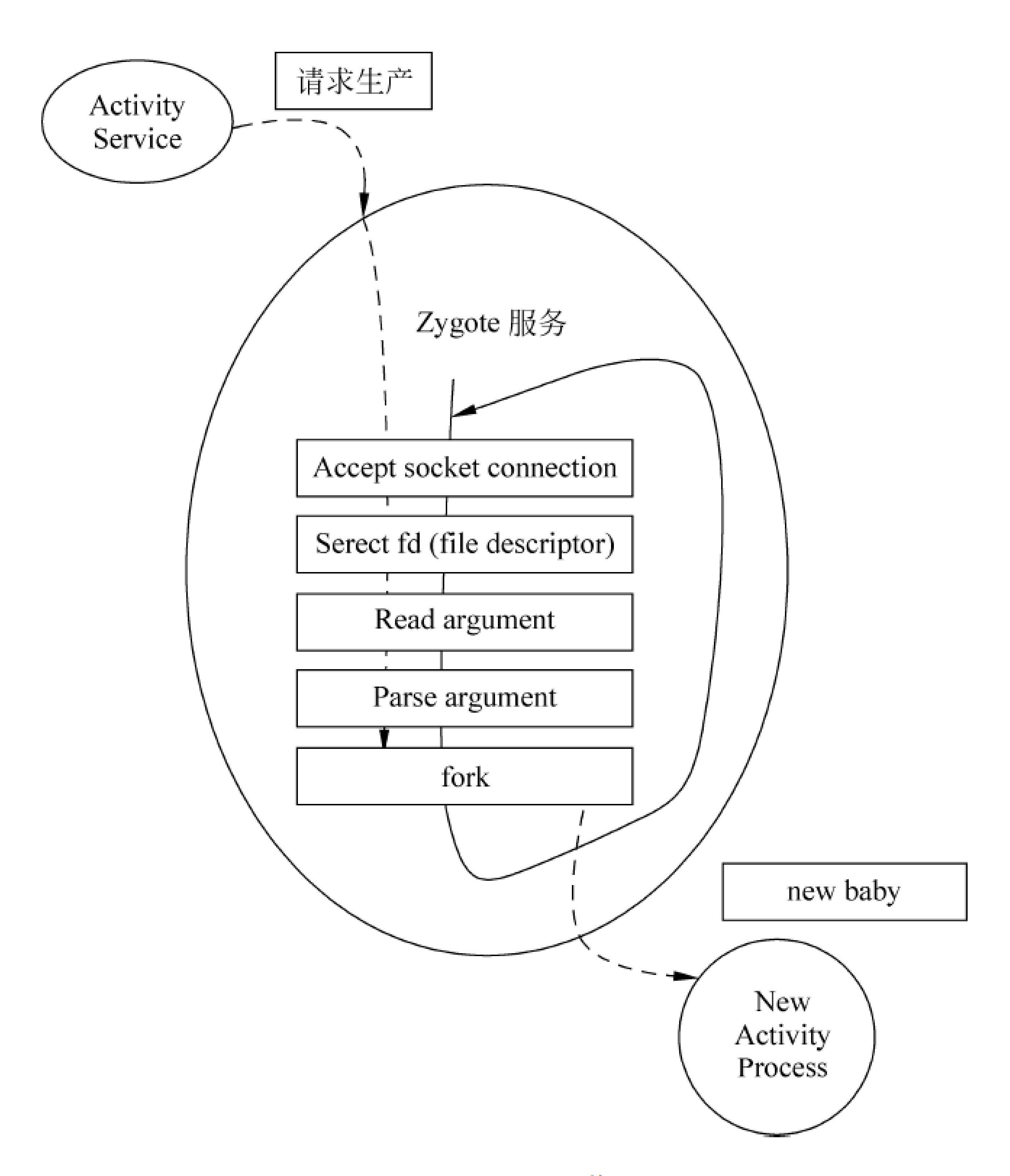


图 6-4　Zygote 工作原理

查看 Zygote 的源代码，可以了解 Zygote 的组成及其调用结构，Zygote 的核心代码包括如下几个文件：

1）Zygote. java

提供访问 Dalvik Zygote 的接口。主要是包装 Linux 系统的 Fork，以建立一个新的 VM 实例进程。

2）ZygoteConnection. java

Zygote 的套接口连接管理及其参数解析。其他 Actvitiy 建立进程请求是通过套接口发送命令参数给 Zygote。

3）ZygoteInit. java

Zygote 的 main 函数入口。

Zygote 系统代码层次调用关系如下：

```
main()
startSystemServer() …
runSelectLoopMode()
Accept socket connection
Conntecion. RunOnce()
Read argument
folkAndSpecialize
```

其中，folkAndSpecialize 是本地方法 Dalvik_dalvik_system_Zygote_forkAndSpecialize。这个本地方法中包含在 dalvik_system_Zygote.c 文件中，其中有如下代码：

```
const DalvikNativeMethod dvm_dalvik_system_Zygote[] = {
  { "fork",            "()I",
    Dalvik_dalvik_system_Zygote_fork},
  { "forkAndSpecialize",           "(II[II[[I)I",
    Dalvik_dalvik_system_Zygote_forkAndSpecialize},
  { "forkSystemServer",           "(II[II[[I)I",
    Dalvik_dalvik_system_Zygote_forkSystemServer},
  { NULL, NULL, NULL},
};
```

Android 使用了 Linux 的 fork 机制。在 Linux 中 fork 是很高效的。一个 Android 的应用实际上是一个 Linux 进程，所谓进程具备下面几个要素：

- 要有一段程序供该进程运行，程序是可以被多个进程共享的。
- 进程专用的系统堆栈空间。
- 进程控制块，在 Linux 中的具体实现是 task_struct。
- 有独立的存储空间。

fork 创造的子进程复制了父亲进程的资源，包括内存的内容 task_struct 内容，在复制过程中，子进程复制了父进程的 task_struct，系统堆栈空间和页面表，而当子进程改变了父进程的变量时，会通过 copy_on_write 的手段为所涉及的页面建立一个新的副本。所以只有子进程有改变变量时，子进程才新建了一个页面复制原来页面的内容，基本资源的复制是必需的，整体看上去就像是父进程的独立存储空间也复制了一遍。

3. Dalvik 虚拟机

Dalvik 是 Google 公司自己设计用于 Android 平台的 Java 虚拟机。Dalvik 虚拟机是 Google 等厂商合作开发的 Android 移动设备平台的核心组成部分之一。它可以支持已转换为 .dex(即 Dalvik Executable)格式的 Java 应用程序的运行，.dex 格式是专为 Dalvik 设计的一种压缩格式，适合内存和处理器速度有限的系统。Dalvik 经过优化，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个 Dalvik 应用作为一个独立的 Linux 进程执行。独立的进程可以防止在虚拟机崩溃时所有程序都被关闭。

Dalvik 和标准 Java 虚拟机(JVM)的首要差别是：

Dalvik 基于寄存器，而 JVM 基于栈。基于寄存器的虚拟机对于更大的程序来说，在它们编译的时候，花费的时间更短。

Dalvik 和 Java 运行环境的区别有如下几点：

- Dalvik 主要是完成对象生命周期管理、堆栈管理、线程管理、安全和异常管理以及垃圾回收等重要功能。
- Dalvik 负责进程隔离和线程管理，每一个 Android 应用在底层都会对应一个独立的 Dalvik 虚拟机实例，其代码在虚拟机的解释下得以执行。
- 不同于 Java 虚拟机运行 Java 字节码，Dalvik 虚拟机运行的是其专有的文件格式 dex。

- dex 文件格式可以减少整体文件尺寸，提高 I/O 操作的类查找速度。
- odex 是为了在运行过程中进一步提高性能，对 dex 文件进一步优化。
- 所有的 Android 应用的线程都对应一个 Linux 线程，虚拟机因而可以更多地依赖操作系统的线程调度和管理机制。
- 有一个特殊的虚拟机进程 Zygote，它是虚拟机实例的孵化器。它在系统启动的时候就会产生，会完成虚拟机的初始化、库的加载、预制类库和初始化的操作。如果系统需要一个新的虚拟机实例，它会迅速复制自身，以最快的数据提供给系统。对于一些只读的系统库，所有虚拟机实例都和 Zygote 共享一块内存区域。

6.2.2 Android 线程

虽然可能将应用程序限制在一个进程中，但有时候会需要衍生一个线程做一些后台工作。因为用户界面必须很快响应用户的操作，所以活动寄宿的线程不应该做一些耗时的操作，如网络下载。任何不能在短时间完成的操作都应该分配到其他线程。

线程在代码中是用标准的 Java 线程对象创建的，Android 提供了一些方便的类来管理线程——Looper 用于在线程中运行消息循环、Handler 用户处理消息、HandlerThread 用户设置一个消息循环的线程。

Looper 类在线程中运行消息循环。线程默认没有消息循环，可以在线程中调用 prepare()创建一个运行循环；然后调用 loop()处理消息直到循环结束。大部分消息循环交互是通过 Handler 类。下面是一个典型的执行一个 Looper 线程的例子，分别使用 prepare()和 loop()创建一个初始的 Handler 与 Looper 交互：

```
class LooperThread extends Thread{
      public Handler mHandler;

      public void run(){
          Looper.prepare();

          mHandler = new Handler(){
              public void handleMessage(Message msg){
                  // process incoming messages here
              }
          };

            Looper.loop();
      }
}
```

每个进程包含一个或多个线程。在多数情况下，Android 避免在进程里创建额外的线程，以保持应用程序单线程，除非它创建自己的线程。一个重要的结果就是所有对活动 Activity、广播接收器 BroadcastReceiver 以及服务 Service 实例的调用都是由这个进程的主线程创建的。

注意新的线程并不会为每个活动、广播接收器、服务或者内容提供器(ContentProvider)实例而创建：这些应用程序的组件在进程里被实例化(除非另有说明，都在同一个进程处理)，

实际上是进程的主线程。这说明当被系统调用时，没有哪个组件（包括服务）会进行远程或者阻塞操作（就像网络调用或者计算循环），因为这将阻止进程中的所有其他组件。可以使用标准的线程类 Thread 或者 Android 的 HandlerThread 便捷类去对其他线程执行远程操作。

这里有一些关于这个线程规则的重要的例外：

- 对 IBinder 或者 IBinder 实现的接口的调用由调用线程或本地进程的线程池（如果该呼叫来自其他进程）分发，而不是它们的进程的主线程。在特殊情况下，一个服务的 IBinder 可以这样调用（尽管调用服务里的方法已经在主线程中完成）。这意味着 IBinder 接口的实现必须要有一种线程安全的方法，这样任意线程才能同时访问它。
- 对 ContentProvider 主要方法的调用由调用线程或者主线程分发，如同 IBinder 一样。被指定的方法在内容提供器的类中有记录。这意味着实现这些方法必须要有一种线程安全的模式，这样任意其他线程可以同时访问它。
- 视图及其子类中的调用由正在运行着视图的线程产生。通常情况下，这会被作为进程的主线程，如果创建一个线程并显示一个窗口，那么继承的窗口视图将从那个线程中启动。

6.2.3 Android 的单线程模型

当一个程序第一次启动时，Android 会同时启动一个对应的主线程（Main Thread），主线程主要负责处理与 UI 相关的事件，如用户的按键事件、用户接触屏幕的事件以及屏幕绘图事件，并把相关的事件分发到对应的组件进行处理。所以主线程通常又被叫做 UI 线程。在开发 Android 应用时必须遵守单线程模型的原则：Android UI 操作并不是线程安全的并且这些操作必须在 UI 线程中执行。如果在非 UI 线程中直接操作 UI 线程，会抛出 android. view. ViewRoot $ CalledFromWrongThreadException：Only the original thread that created a view hierarchy can touch its views，这与普通的 Java 程序不同。

由于 UI 线程负责事件的监听和绘图，因此，必须保证 UI 线程能够随时响应用户的需求，UI 线程中的操作应该向中断事件那样短小，费时的操作（如网络连接）需要另开线程；否则，如果 UI 线程超过 5s 没有响应用户请求，会弹出 Force Close 对话框提醒用户终止应用程序。

6.2.4 Android 多线程

如果在新开的线程中需要对 UI 进行设定，就可能违反单线程模型，因此 Android 采用一种稍微复杂的 Message Queue 机制来进行线程间通信。

Message Queue 机制需要通过 handle 来自己管理线程类，如果业务稍微复杂，代码看起来就比较混乱，因此 Android 还提供了 AsyncTask 类来解决此问题。

接下来就对这两种方式进行简要介绍，在 6.3 节中将会进一步对其进行介绍。

1. 消息队列

Message Queue 即消息队列，用来存放通过 Handler 发布的消息。Android 在第一次启动程序时会默认会为 UI thread 创建一个关联的消息队列，可以通过 Looper. myQueue()

得到当前线程的消息队列，用来管理程序的一些上层组件、activities、broadcast receivers 等。可以在自己的子线程中创建 Handler 与 UI thread 通信。

通过 Handler 可以发布或者处理一个消息或者是一个 Runnable 的实例。每个 Handler 都会与唯一的一个线程以及该线程的消息队列管理。

Looper 扮演着 Handler 和消息队列之间通信桥梁的角色。程序组件首先通过 Handler 把消息传递给 Looper，Looper 把消息放入队列。Looper 也把消息队列中的消息广播给所有的 Handler，Handler 接收到消息后调用 handleMessage 进行处理。

下面通过一个示例来介绍消息队列，示例项目名称为 MessageQueueDemo，需要读者注意的是，这个示例附带的源码一开始是有错误的，这个错误就是为了说明在非 UI 线程中尝试更新 UI 组件将会引发的错误，在之后将会应用 MessageQueue 对这个错误进行修正。示例 Activity 代码如下：

```
public class MessageQueueActivity extends Activity implements OnClickListener{
  MessageHandler messageHandler;
  @Override
  public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    Button button = (Button) findViewById(R.id.button1);
    button.setOnClickListener(this);
    //得到当前线程的 Looper 实例,由于当前线程是 UI 线程,
    //因此也可以通过 Looper.getMainLooper()得到
    Looper looper = Looper.myLooper();
    //此处甚至可以不需要设置 Looper,因为 Handler 默认就使用当前线程的 Looper
    messageHandler = new MessageHandler(looper);
  }

  public void onClick(View v){
    new Thread(){
      public void run(){
        //Message message = Message.obtain();
        //message.obj = "子线程更新标题显示";
        //messageHandler.sendMessage(message);              //发送消息
        setTitle("子线程更新标题显示");
      }
    }.start();
  }

  class MessageHandler extends Handler{
    public MessageHandler(Looper looper){
      super(looper);
  }

    @Override
    public void handleMessage(Message msg){
      setTitle((String) msg.obj);
    }
  }
}
```

其中被注释掉的 3 行代码（第 19～21 行）是非 UI 线程向消息队列发出消息的操作，当不使用这种方式而是直接在新的非 UI 线程中试图执行更改 UI 的操作时（第 22 行），将会抛出前面所提到的异常，并弹出 Force Close 窗口，如图 6-5 所示。

```
FATAL EXCEPTION: Thread-10
android.view.ViewRoot$CalledFromWrongThreadException: Only the original thread that created a
    at android.view.ViewRoot.checkThread(ViewRoot.java:2932)
    at android.view.ViewRoot.invalidateChild(ViewRoot.java:642)
    at android.view.ViewRoot.invalidateChildInParent(ViewRoot.java:668)
    at android.view.ViewGroup.invalidateChild(ViewGroup.java:2511)
    at android.view.View.invalidate(View.java:5279)
    at android.widget.TextView.checkForRelayout(TextView.java:5507)
    at android.widget.TextView.setText(TextView.java:2724)
    at android.widget.TextView.setText(TextView.java:2592)
    at android.widget.TextView.setText(TextView.java:2567)
    at com.android.internal.policy.impl.PhoneWindow.setTitle(PhoneWindow.java:278)
    at android.app.Activity.onTitleChanged(Activity.java:3591)
    at android.app.Activity.setTitle(Activity.java:3557)
    at com.android.example.message.MessageQueueActivity$1.run(MessageQueueActivity.java:30)
```

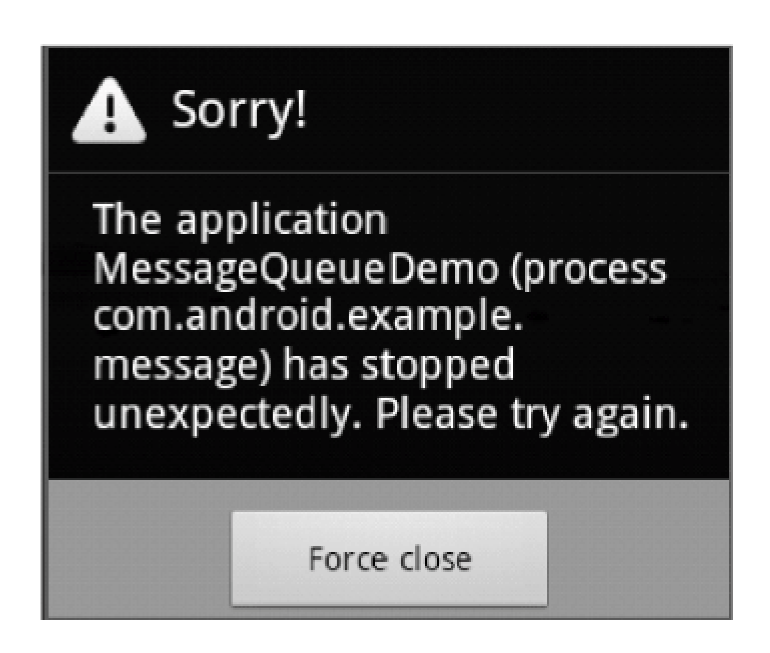


图 6-5　程序出现异常

为了正确地对 UI 进行更改，就需要使用到消息队列。只需要将上述代码中的第 19～21 行取消掉注释，并且删除第 22 行的代码，程序就可以正常运行了。此例中是通过响应屏幕上按钮的单击事件，在被单击时开启一个新的线程，并在该线程内对 UI 进行操作，示例效果如图 6-6 所示。



图 6-6　单击按钮前后的界面（标题发生了变化）

2. AsyncTask

使用 Handler 来处理消息队列的方式来实现非 UI 线程对 UI 的更新，对于开发人员来说，如果业务逻辑比较复杂，代码会显得比较杂乱，因此 Android 还提供了另外一种方式来方便使用 UI 线程，即 AsyncTask 类。

AsyncTask 类允许开发人员在不需要管理线程和 Handler 的情形下进行一些后台操作并将结果反映至 UI 线程。简单地说，就是将该部分任务直接托管给 Android。因此在一

定程度上方便了开发人员。

AsyncTask 必须通过继承该类来使用，子类必须至少重写一个 doInBackground(Params...)方法，通常也会重写 onPostExecute(Result)等几个方法，一个 AsyncTask 对象的执行经过如下 4 个步骤：

- onPreExecute()，该方法将在执行实际的后台操作前被 UI thread 调用。可以在该方法中做一些准备工作，如在界面上显示一个进度条。
- doInBackground(Params...)，将在 onPreExecute 方法执行后马上执行，该方法运行在后台线程中。这里将主要负责执行那些很耗时的后台计算工作。可以调用 publishProgress 方法来更新实时的任务进度。该方法是抽象方法。
- onProgressUpdate(Progress...)，在 publishProgress 方法被调用后，UI thread 将调用这个方法从而在界面上展示任务的进展情况，例如通过一个进度条进行展示。
- onPostExecute(Result)，在 doInBackground 执行完成后，onPostExecute 方法将被 UI thread 调用，后台的计算结果将通过该方法传递到 UI thread。

使用 AsyncTask 时需要遵循以下几点规则：

- Task 的实例必须在 UI thread 中创建。
- execute 方法必须在 UI thread 中调用。
- 不要手动调用这些方法，只调用 execute 即可。
- 该 task 只能被执行一次，否则多次调用时将会出现异常。

下面通过一个示例来介绍 AsyncTask 的使用，示例项目名称为 AsyncTaskDemo。这个示例实现的功能是：使用 AsyncTask 方式在后台获取天气信息，完成后更新 UI 元素使天气信息被显示。

该 Project 主要包含了 3 个类：

- AsyncTaskActivity 是启动 Activity，主要用于将通过 AsyncTask 所获取的数据显示到用户界面中。
- CurrentCondition 类是用于存放从 xml 文件中解析出来的天气数据。
- DomXMLReader 类用于解析由 Google 天气 API 所返回的 xml 格式数据。

由于本例中主要介绍的是 AsyncTask 的使用方法，因此有关如何从互联网上获取到实时的天气预报信息，这里将不进行介绍，8.4 节将对此进行详细介绍。

在 AsyncTaskActivity 中有关 AsyncTask 使用的代码段如下：

```
class GetWeatherTask extends AsyncTask<String, Integer, String> {

  @Override
  protected String doInBackground(String... params) {
    String city = params[0];

    // 调用 Google 天气 API 查询指定城市的当日天气情况
    return getWeatherByCity(city);
  }

  protected void onPostExecute(String result) {
    // 把 doInBackground 处理的结果即天气信息显示在 TextView 上
```

```
        tvWeatherInfo.setText(result);
        icon.setImageBitmap(bm);
    }
}
```

该示例的运行效果如图 6-7 所示，读者在运行该示例的时候可以发现，当 AsyncTask 在后台通过网络获取天气信息时并不会阻塞用户界面，当后台获取天气信息完毕后，信息会自动地显示到界面上。

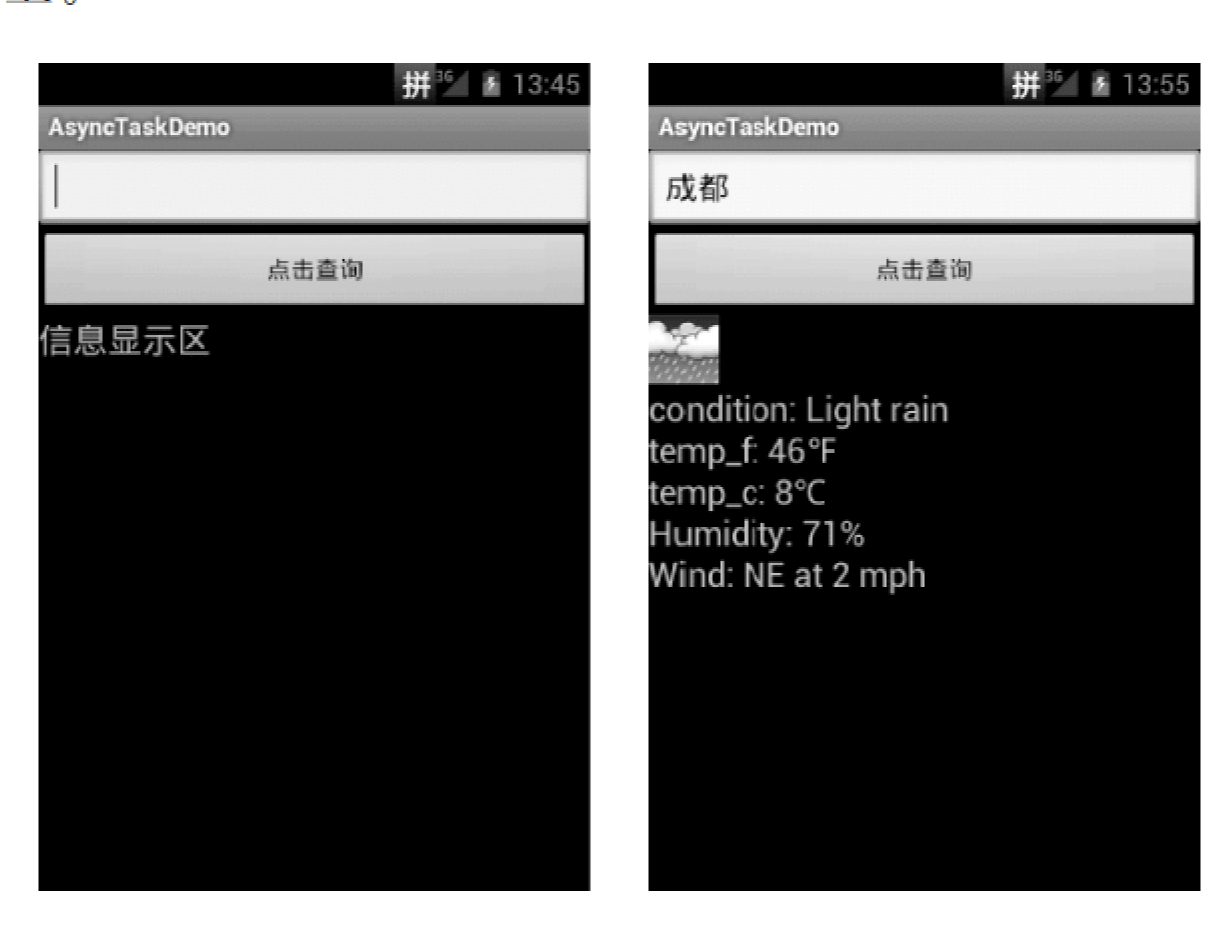


图 6-7 AsyncTaskDemo 运行效果

6.3 消息机制

6.3.1 消息机制的引入

为什么要引入消息机制呢？在前面使用 Message Queue 时已经提到了，这是由于在 Android 的安全机制下，在非 UI 线程中不能够直接对 UI 进行操作。在 Android 应用程序启动的时候，会首先启动一个主线程（即 UI 线程）。当有时需要进行一些耗时运算时，如果这些运算都放在主线程中，那么主线程就会阻塞，这样将会造成用户界面失去响应的现象。因此，在非 UI 线程中需要操作 UI 时，就需要借助 Android 的消息机制来与 UI 线程进行通信，从而达到更新 UI 的目的。同理，在其他的线程中，也可以使用消息机制来实现线程的异步操作。

6.3.2 Android 消息机制的构成

构成 Android 消息机制的主要有如下几个类：

- Message——消息对象，顾名思义就是记录消息信息的类。
- MessageQueue——消息队列，用来存放 Message 对象的数据结构，按照“先进先出”的原则存放消息。
- Looper——MessageQueue 的管理者，使消息队列能够被有序处理。

- Handler——消息的处理者。

下面对这 4 个类进行更加详细的介绍。

1. Message

Message 这个类有如下几个比较重要的字段：

- arg1 和 arg2——可以使用两个字段用来存放所需要传递的整型值，在 Service 中，可以用来存放 Service 的 ID。
- obj——该字段是 Object 类型，可以让该字段传递某个多项到消息的接收者中。
- what——这个字段可以说是消息的标志，在消息处理中，可以根据这个字段的不同的值进行不同的处理，类似于在处理 Button 事件时，通过 switch(v.getId())判断是单击了哪个按钮。

在使用 Message 时，可以通过 new Message()创建一个 Message 实例，但是 Android 推荐通过 Message.obtain()或者 Handler.obtainMessage()获取 Message 对象。这并不一定是直接创建一个新的实例，而是先从消息池中看有没有可用的 Message 实例，存在则直接取出并返回这个实例。反之，如果消息池中没有可用的 Message 实例，则根据给定的参数 new 一个新 Message 对象。通过分析源码可得知，Android 系统默认情况下在消息池中实例化 10 个 Message 对象。

2. MessageQueue

MessageQueue(消息队列)用来存放 Message 对象的数据结构，按照“先进先出”的原则存放消息。存放并非实际意义的保存，而是将 Message 对象以链表的方式串联起来的。MessageQueue 对象不需要自己创建，而是有 Looper 对象对其进行管理，一个线程最多只可以拥有一个 MessageQueue。可以通过 Looper.myQueue()获取当前线程中的 MessageQueue。

3. Looper

Looper，MessageQueue 的管理者。在一个线程中，如果存在 Looper 对象，则必定存在 MessageQueue 对象，并且只存在一个 Looper 对象和一个 MessageQueue 对象。在 Android 系统中，除了主线程有默认的 Looper 对象，其他线程默认是没有 Looper 对象的。如果想让线程拥有 Looper 对象，首先应调用 Looper.prepare()方法，然后再调用 Looper.loop()方法。典型的用法如下所示：

```
class LooperThread extends Thread
{
    public Handler mHandler;
    public void run()
    {
        Looper.prepare();
        //其他需要处理的操作
        Looper.loop();
    }
}
```

如果线程中存在 Looper 对象，则可以通过 Looper. myLooper()来获取，此外还可以通过 Looper. getMainLooper()获取当前应用的主线程的 Looper 对象。在这个地方有一点需要注意：假如 Looper 对象位于应用程序主线程中，则 Looper. myLooper()和 Looper. getMainLooper()获取的是同一个对象。

4. Handler

Handler，消息的处理者。通过 Handler 对象可以封装 Message 对象，然后通过 sendMessage(msg)把 Message 对象添加到 MessageQueue 中；当 MessageQueue 循环到该 Message 时，就会调用该 Message 对象对应的 handler 对象的 handleMessage()方法对其进行处理。由于是在 handleMessage()方法中处理消息，因此通常会编写一个类继承自 Handler，然后在 handleMessage()中根据获取的消息来执行需要的操作。Handler 将每一个消息发送到队列里面，遵循先进先出原则。发送消息并不会阻塞线程，而接收线程会阻塞线程，这是因为 Android 的 Handler 机制，当 Handler 处理完一个 Message 对象才会接着去取下面一个消息进行处理，如图 6-8 所示。

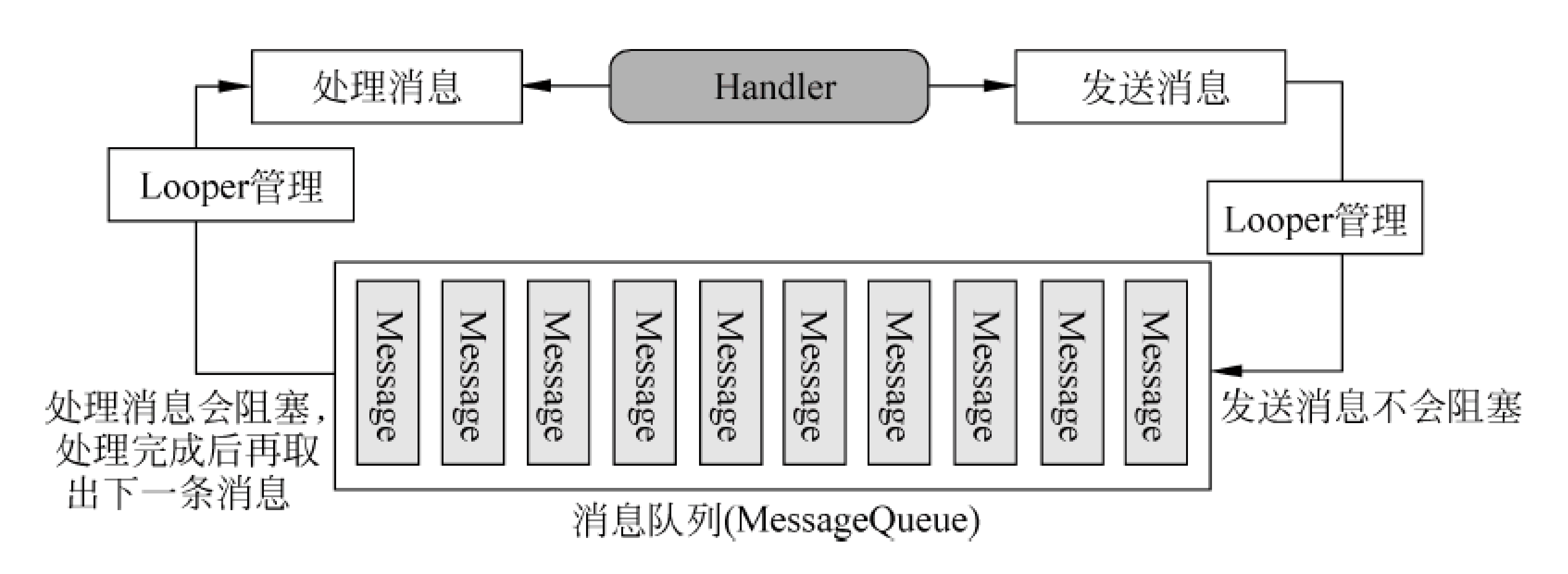


图 6-8 Android 消息机制图解

Android 里并没有 Global 的 Message Queue 数据结构，因此，不同 APK 中的对象不能通过 Massage Queue 来交换消息(Message)。线程 A 的 Handler 对象可以传递消息给其他线程，让其他线程 B 或 C 能够发送消息给线程 A(存于 A 的 Message Queue 中)。线程 A 的 Message Queue 中的消息，只有线程 A 所属的对象可以处理。

6.3.3 消息机制示例

下面通过一个示例并结合 Android 源码来了解消息机制的工作流程，示例项目名称为 MessageDemo。

本例是一个用于处理消息的服务类(Service)的实现，在该 Service 中新建了一个专用于处理消息的线程(HandlerThread)以防止耗时操作阻塞主线程，HandlerThread 类是 Thread 的子类，该类运行时将会创建 looper 对象，使用该类可省去人工再创建 Looper 的步骤。在创建了该线程后调用其 start()方法开始执行线程，之后可以通过 thread 的 getLooper()方法获取到该线程的 Looper，将该 Looper 作为参数传入 Handler，就建立起了 handler、looper 及 messageQueue 三者的联系，就可以实现利用消息机制达到线程间通信的效果。

首先看一看本例中服务类 MessageService 的 onCreate()代码：

```
@Override
public void onCreate()
{
  Log.i(TAG, "MessageService --> onCreate()");

  HandlerThread thread = new HandlerThread("MessageDemoThread",
      Process.THREAD_PRIORITY_BACKGROUND);
  thread.start();
  looper = thread.getLooper();
  handler = new ServiceHandler(looper);
}
```

重写了 Service 的 onStartCommand()方法，使得服务在启动时自动发送需要处理的消息至消息队列。

```
@Override
public int onStartCommand(Intent intent, int flags, int startId)
{
  Log.i(TAG, "MessageService--> onStartCommand()");
  Message msg = handler.obtainMessage();
  //arg1 保存线程的 ID,在 handleMessage()方法中
  //可以通过 stopSelf(startId)方法,停止服务
  msg.arg1 = startId;
  msg.what = MESSAGE_CODE;
  Date date = new Date();
  msg.obj = date;
  handler.sendMessage(msg);
  return START_STICKY;
}
```

ServiceHandler 的代码如下，这个内部类继承自 Handler，通过重写 handleMessage()方法来实现需要的功能，此处仅仅是将消息中所携带的时间信息输出到 Logcat 中：

```
private final class ServiceHandler extends Handler
{
  public ServiceHandler(Looper looper)
  {
  super(looper);
  }

  @Override
  public void handleMessage(Message msg)
  {
      switch (msg.what)
      {
      case MESSAGE_CODE:
        Log.i(TAG, "The obj field of msg:" + msg.obj);
```

```
                break;
            default:
                break;
            }
            stopSelf(msg.arg1);
        }
    }
```

运行该示例，在 LogCat 中将会得到如图 6-9 所示的日志信息：

tag	Message
MessageService	MessageService-->onCreate()
MessageService	MessageService-->onStartCommand()
MessageService	The obj field of msg:Tue Sep 06 07:33:20 GMT+00:00 2011
MessageService	MessageService-->onDestroy()

图 6-9　LogCat 日志信息

日志说明了程序执行的流程，下面再跟着代码来分析一下 Android 的消息机制到底是如何工作的。

启动服务时将会首先调用 MessageService 类的 onCreate()方法，在该方法中新建了一个 HandlerThread 对象，并提供了线程的名字和优先级。

紧接着调用了 start()方法，执行该方法将会调用 HandlerThread 对象的 run()方法，run()方法的源代码如下：

```
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
    mLooper = Looper.myLooper();
    notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

可见在 HandlerThread 中系统自动创建了线程的 Looper 对象，并同时调用了 Looper 的 loop()方法：

```
public static final void loop(){
   Looper me = myLooper();
   MessageQueue queue = me.mQueue;
   while (true){
      Message msg = queue.next();              // might block
      //if (!me.mRun){
      //  break;
      //}
```

```
            if (msg != null) {
                if (msg.target == null) {
                  // No target is a magic identifier for the quit message.
                  return;
                }
                if (me.mLogging! = null) me.mLogging.println(
                      ">>>>> Dispatching to " + msg.target + " "
                      + msg.callback + ": " + msg.what
                      );
               msg.target.dispatchMessage(msg);
               if (me.mLogging! = null) me.mLogging.println(
                      "<<<<< Finished to " + msg.target + " "
                      + msg.callback);
               msg.recycle();
            }
        }
    }
```

loop()方法就是一个无限循环,它不停地从消息队列中取出消息,然后交给 handler 处理并回收消息对象。

start()方法执行完成后,线程的消息循环 looper 则已经进入了正常工作的状态,这时就可以可以获取线程的 Looper 对象,然后新建一个 ServiceHandler 对象,把 Looper 对象传到 ServiceHandler 构造函数中将使 handler、looper 和 messageQueue 三者建立联系。如此一来,就建立起了如图 6-8 所示的消息机制的具体实现。

当 onCreate()方法执行完成后会调用 Service 的 onStart()方法,进一步调用 onStartCommand()方法。

在 onStartCommand()方法中,首先会从消息池中获取一个 Message 实例,然后给 Message 对象的 arg1、what、obj 3 个字段赋值。紧接着调用 sendMessage(msg)方法,分析 Android 源码可以发现,该方法将会调用 sendMessageDelayed(msg, 0)方法,而 sendMessageDelayed()方法又会调用 sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis)方法,在该方法中注意到代码"msg.target = this",即 msg 的 target 指向了 this,而 this 就是 ServiceHandler 对象,因此 msg 的 target 字段指向了 ServiceHandler 对象,同时该方法又调用 MessageQueue 的 enqueueMessage(msg, uptimeMillis)方法。enqueueMessage(msg, uptimeMillis)——顾名思义,即让消息入队,其代码如下:

```
    final boolean enqueueMessage(Message msg, long when) {
        if (msg.when != 0) {
            throw new AndroidRuntimeException(msg
                  + " This message is already in use.");
        }
         if (msg.target == null && !mQuitAllowed) {
            throw new RuntimeException("Main thread not allowed to quit");
        }
        synchronized (this) {
            if (mQuiting) {
```

```
                RuntimeException e = new RuntimeException(
                    msg.target + " sending message to a Handler on a dead thread");
                Log.w("MessageQueue", e.getMessage(), e);
                return false;
            } else if (msg.target == null){
                 mQuiting = true;
            }
            msg.when = when;
            //Log.d("MessageQueue", "Enqueing: " + msg);
            Message p = mMessages;
            if (p == null || when == 0 || when < p.when) {
                msg.next = p;
                mMessages = msg;
                this.notify();
            } else{
                Message prev = null;
                while (p != null && p.when <= when) {
                   prev = p;
                   p = p.next;
                }
                msg.next = prev.next;
                prev.next = msg;
                this.notify();
            }
        }
        return true;
    }
```

dispatchMessage()：用于调度消息的方法。

```
    public void dispatchMessage(Message msg) {
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
            if (mCallback != null) {
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
            handleMessage(msg);
        }
    }
```

该方法首先判断 callback 是否为空，当 handler 不使用回调函数处理消息时，callback 字段为空，所以会执行 handleMessage()方法，也就是在 ServiceHandler 类中重写的方法。在该方法将根据 what 字段的值判断执行哪段代码。

至此，一个 Message 经由 Handler 的发送，MessageQueue 的入队，Looper 的抽取，又再一次地回到 Handler 的怀抱中。而绕的这一圈，正好将同步操作变成了异步操作。

新消息的入队过程如图 6-10 所示。

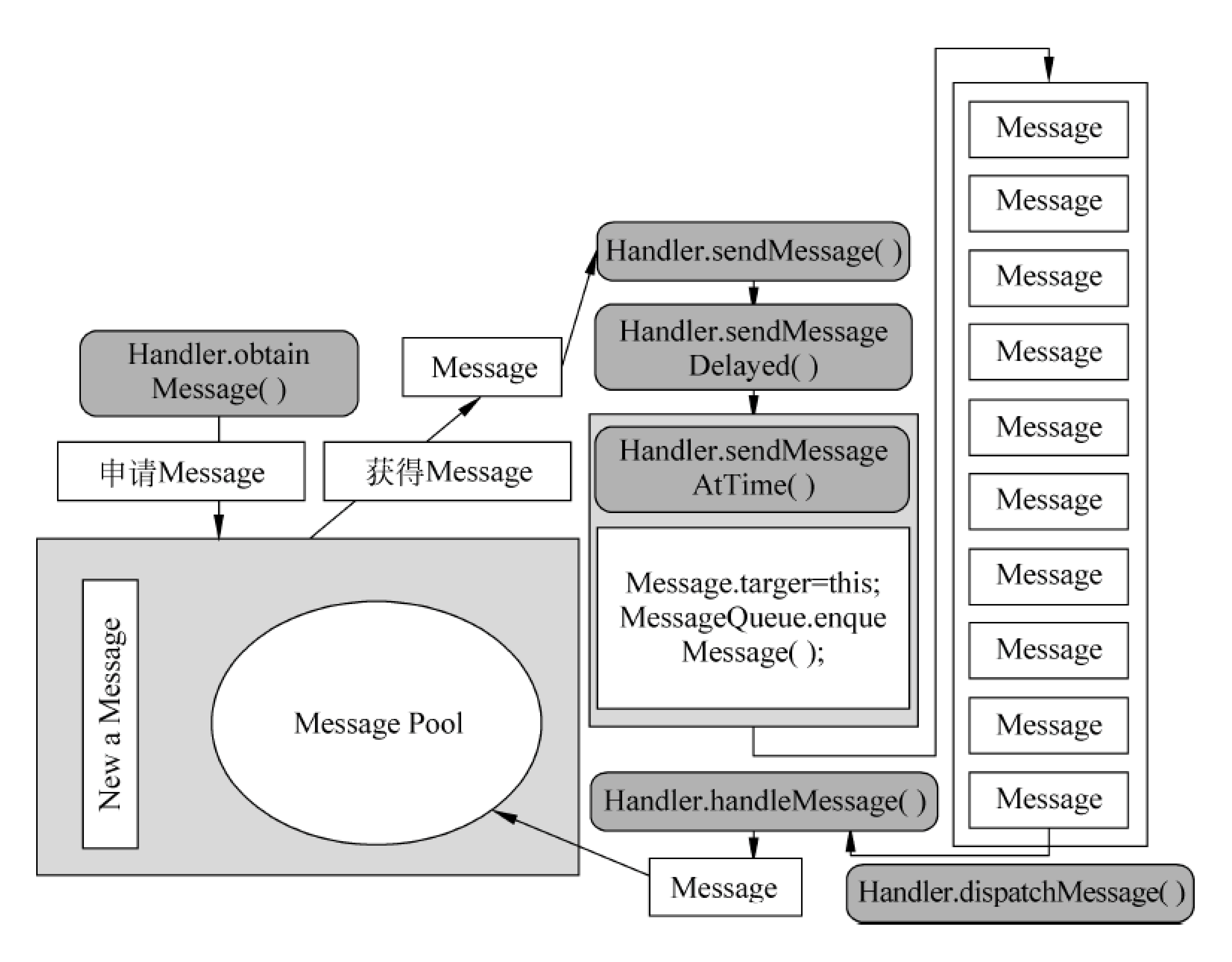


图 6-10　消息的入队过程

6.4　进程间通信

6.4.1　Intent

Android 对于上层的应用开发屏蔽了进程的概念，换而言之的是通过四大组件——<activity>、<service>、<receiver>、<provider>来取代进程的概念，因此 Android 进程间的通信可以理解为这四大组件之间的通信。

组件间通信的核心机制是 Intent，Android 的组件和进程间通信都建立在 Intent 消息基础之上。通过 Intent 可以开启一个 Activity 或 Service，无论这个 Activity 或 Service 是属于当前应用还是其他应用都可以，通过这种方式就实现了一种简单的进程间通信。

Intent 是一个将要执行的动作的抽象的描述，一般来说是作为参数来使用，由 Intent 来协助完成 Android 各个组件之间的通信。比如说调用 startActivity()来启动一个 activity，或者由 broadcastIntent()来传递给所有感兴趣的 BroadcastReceiver，再或者由 startService()/bindservice()来启动一个后台的 service。因此，Intent 主要是用来启动其他的 activity 或者 service，可以将 Intent 理解成各组件之间的链接器。

简单地说，Intent 就是一种消息，它包含了两个重要的内容：

(1) 消息的目的地，即这个消息是发给哪个组件的。

(2) 消息所携带的数据内容，即需要传递给目标的数据。

例如，可以通过如下方式启动一个 service 组件：

```
//用于启动对应 Service 的 intent 对象
Intent intent = new Intent(context, Target.class);
```

```
Bundle bundle = new Bundle();
int op = -1;
bundle.putInt("op", op);                    //将 op 存放于 bundle 中
intent.putExtras(bundle);                   //将 bundle 绑定到 intent
context.startService(intent);               //使用 intent 启动服务
```

上面的代码首先构造了一个 Intent 对象，并指定了这个 Intent 的目的地即类 Target，然后使用 Bundle 作为数据的封装传递了一个值为-1 的整型数据 op。最后通过 startService()方法启动对应服务。

Intent 的消息目的地分为两种模式：一种是显式的；另一种是隐式的。上面的例子中使用的就是显式的模式。该种模式下通过已知的同应用程序下的类名来启动服务，该类必须属于同一个应用程序。

显式和隐式 Intent 的关系和区别如下：

（1）显式 Intent——直接指定消息目的地，只适合同一进程内的不同组件之间通信。

使用方式：

```
new Intent(this,Target.class)
```

显式消息直接指定消息目的地组件的类元信息(Target.class)，这种模式的消息由于已经确切知道了消息目标的确切信息，所以只适用于同一进程内的不同组件之间通信，例如打开一个子窗体，和同一进程中的 service 通信的操作。

（2）隐式 Intent——在 AndroidMainifest.xml 中注册，一般用于跨进程通信。

使用方式：

```
new Intent(String action)
```

隐式消息没有确定的消息目的地，除了数据外，隐式消息只是包含了一些用于表征消息特征的描述字段。而一些需要收到这种特定特征消息的某个程序中的某个组件，需要通过在其所在程序的 AndroidMainifest.xml 中注册一种被称为 intent-filter 的消息特征筛选器，然后 Android 系统会按照一定的匹配规则来匹配发出的消息特征和所有拥有响应这种特征的 intent-filter 的组件(无论是同一进程内的组件还是不同程序中的组件)，匹配到的组件就会接收到相应的消息。

1. 隐式 Intent 用法

首先需要提供调用一方在 AndroidManifest.xml 进行注册：

```
<service android:enabled="true" android:name=".MusicService">
<intent-filter>
<action android:name="com.android.ServiceDemo.musicService" />
</intent-filter>
</service>
```

这表示该 Service 能够接收到并处理 action 为 com.android…musicService 的消息。

之后就可以通过隐式 Intent 的方式来启动该服务了，该服务可以是其他应用程序(其他进程)中的服务，由 Android 系统负责唤起服务执行：

```
//用于启动对应 Service 的 intent 对象
Intent intent = new Intent("com.android.ServiceDemo.musicService");
Bundle bundle = new Bundle();
int op = -1;
bundle.putInt("op", op);                              //将 op 存放于 bundle 中
intent.putExtras(bundle);                             //将 bundle 绑定到 intent
startService(intent);                                 //使用 intent 启动服务
```

2. Intent 的组成

要在不同的组件之间传递信息，就需要通过 Intent 来携带数据，包括如下几种：

- action——用来指明要实施的动作是什么，可以由用户自定义 action，系统也提供了一些常用的 action，例如 ACTION_VIEW，ACTION_EDIT 等，这些常用的 action 可以在 android.content.intent 类找到定义。
- Data——要事实的具体的数据，一般由 Uri 指定，例如：

```
ACTION_VIEW content://contacts/1      //显示 identifier 为 1 的联系人的信息
ACTION_DIAL content://contacts/1      //给这个联系人打电话
```

- Category(类别)：Category 同样是一个字符串，从字面上理解就是"消息的分类特征"。这个选项指定了将要执行的这个 action 的分类信息。
- Type(数据类型)：显式指定 Intent 的数据类型(MIME)。一般 Intent 的数据类型能够根据数据本身进行判定，但是通过设置这个属性，可以强制采用显式指定的类型而不再进行推导。
- Component(组件)：指定 Intent 的目标组件的类名称。通常 Android 会根据 Intent 中包含的其他属性的信息，比如 action、data/type、category 进行查找，最终找到一个与之匹配的目标组件。但是，如果 component 这个属性有指定的话，将直接使用它指定的组件，而不再执行上述查找过程。指定了这个属性以后，Intent 的其他所有属性都是可选的。
- Extras(附加信息)，是其他所有附加信息的集合。使用 extras 可以为组件提供扩展信息，比如，如果要执行"发送电子邮件"这个动作，可以将电子邮件的标题、正文等保存在 extras 中，传给电子邮件发送组件。

6.4.2 Intent Filter

前面介绍隐式 Intent 使用方法的时候，已经接触到了 Intent Filter 的概念，Intent Filter 通常以子元素标签的形式存在于 AndroidManifest.xml 文件的组件元素下，用于描述该类组件可以接收特定规则的 Intent 消息。

组件为了告诉 Android 自己能响应、处理哪些隐式 Intent 请求，可以声明一个甚至多个 Intent Filter。每个 Intent Filter 描述该组件所能响应 Intent 请求的能力——组件希望接收什么类型的请求行为，什么类型的请求数据等。

Intent Filter 进行匹配时的三要素是 Intent 的动作、数据以及类别(3 种要素不一定同时存在)。实际上，一个隐式 Intent 请求要能够传递给目标组件，必须通过这 3 个方面的检

查。如果任何一方面不匹配，Android 都不会将该隐式 Intent 传递给目标组件。

1. Intent Filter 匹配规则

1）Action 匹配

<intent-filter>元素中可以包括子元素<action>，例如：

```
<intent-filter>
<action android:name="com.example.project.SHOW_CURRENT" />
<action android:name="com.example.project.SHOW_RECENT" />
<action android:name="com.example.project.SHOW_PENDING" />
</intent-filter>
```

一条<intent-filter>元素至少应该包含一个<action>，否则任何 Intent 请求都不能和该<intent-filter>匹配。如果 Intent 请求的 Action 和<intent-filter>中的某一条<action>匹配，那么该 Intent 就通过了这条<intent-filter>的匹配。

如果 Intent 请求或<intent-filter>中没有说明具体的 Action 类型，那么会出现下面两种情况。

（1）如果<intent-filter>中没有包含任何 Action 类型，那么无论什么 Intent 请求都无法和这条<intent-filter>匹配。

（2）反之，如果 Intent 请求中没有设定 Action 类型，那么只要<intent-filter>中包含有 Action 类型，这个 Intent 请求就将顺利地通过<intent-filter>的匹配。

2）Category 匹配

< intent-filter>元素可以包含<category>子元素，例如：

```
<intent-filter>
    <category android:name="android.Intent.Category.DEFAULT"/>
    <category android:name="android.Intent.Category.BROWSABLE"/>
</intent-filter>
```

当 Intent 请求中所有的 Category 与组件中某一个 IntentFilter 的<category>完全匹配时，该 Intent 请求将成功匹配，IntentFilter 中多余的<category>声明并不会导致匹配失败。一个没有指定任何类别匹配的 IntentFilter 仅仅只会匹配没有设置类别的 Intent 请求。

3）data 匹配

data 在< intent-filter>中的描述如下：

```
<intent-filter>
    <data android:type="video/mpeg" android:scheme="http" />
    <data android:type="audio/mpeg" android:scheme="http" />
</intent-filter>
```

元素指定了希望接受的 Intent 请求的数据 URI 和数据类型，URI 被分成 3 部分来进行匹配：scheme、authority 和 path。其中，用 setData()设定的 Intent 请求的 URI 数据类型和 scheme 必须与 IntentFilter 中所指定的一致。若 IntentFilter 中还指定了 authority 或

path,则它们也需要相匹配才会成功匹配。

2. Intent 示例

1）示例一

```
Intent i = new Intent(Intent.ACTION_DIAL,Uri.parse("tel://13800138000"));
startActivity(i);
```

Activity 启动后如图 6-11 所示。

2）示例二

```
Intent intent = new Intent(Intent.ACTION_EDIT,
null);
startActivity(intent);
```

执行此代码的时候,系统就会在程序主配置文件 AndroidMainfest.xml 中寻找<action android:name="android.intent.action.EDIT"/>对应的 Activity,如果对应为多个 activity 就会弹出一个 dailog 选择 Activity。



图 6-11　通过 Intent 启动该拨号程序

6.4.3　Android IPC

有了前面提到的 Intent 这种基于消息的进程内或进程间通信模型,就可以通过 Intent 去开启一个 Service,或者通过 Intent 跳转到另一个 Activity,无论上面的 Service 或 Activity 是在当前进程还是其他进程内,不论是当前应用还是其他应用的 Service 或 Activity,通过 Intent 消息机制都可以进行通信。

然而通过 Intent 消息机制实现的进程间通信仍然不能满足某些方面的需求,例如当 Activity 与 Service(组件与组件)之间的交互不仅仅是简单的 Activity 开启 Service、Activity 之间的跳转,而是需要随时地向其他组件发送控制请求,那么就必须保证 Activity 在 Service 的运行过程中随时可以连接到 Service 并对其进行需要的控制。

因此,Android 还提供了用于进程间交互的 IPC 机制,它采用类似远程方法调用的方案,通过 Android 接口定义语言 AIDL 来定义 IPC 接口,然后在被调用方实现接口,调用方调用接口的本地代理来完成 IPC。这种模型只适用于 Activity 和 Service 间的通信,之所以需要这种模式,就是因为 Activity 除了发送 Intent 去启动 Service 这种方式外,还可能需要能够在 Service 的运行过程中连接到 Service,对 Service 发送一些控制请求。

以音乐播放程序为例,播放服务往往在后台独立运行,以使用户在其他界面时也能听到音乐。同时这个播放服务通常会定义一个控制接口,包含比如播放、暂停、快进等方法,任何时候播放控制界面都能通过 bindService 连接到播放服务,并获取这个接口的包含 IPC 细节的实现代理,并通过这组控制接口方法对其进行控制。

对于音乐播放程序来说，虽然也可以通过 Intent 消息机制来实现对后台 Service 的控制，但是那样会显得比较烦琐，需要在双方约定好 Intent 的解析机制，同时也增加了两者的耦合度。在后面的示例中会分别以这两种方式实现加以比较。

类似于远程方法调用，Android IPC 以 C/S 模式进行访问。首先通过 AIDL 接口文件来定义一个 IPC 接口，并在 Server 端实现 IPC 接口，Client 端调用 IPC 接口的本地代理来使用接口所提供的方法。

由于 IPC 调用是同步的，如果一个 IPC 服务需要超过几毫秒的时间才能完成，就应该避免在 Activity 的主线程中调用该服务，否则 IPC 调用会挂起应用程序导致界面失去响应。在这种情况下，应该考虑调用一个线程来进行 IPC 访问。

两个进程间 IPC 从表面上看起来就像是一个进程进入另一个进程执行代码然后带着执行的结果返回。

Android 的 IPC 机制鼓励开发人员"尽量利用已有功能，利用 IPC 和包含已有功能的程序协作完成一个完整的项目"。

6.4.4 AIDL

1. 什么是 AIDL

为了使其他的应用程序也可以访问本应用程序提供的服务，Android 系统采用了远程过程调用(Remote Procedure Call，RPC)方式来实现。与很多其他的基于 RPC 的解决方案一样，Android 使用一种接口定义语言(Interface Definition Language，IDL)来公开服务的接口，即 AIDL(Android Interface Definition Language)。借助 AIDL 可以快速地生成接口代码，使得在同一个 Android 设备上运行的两个进程之间可以通过内部通信进程进行交互。如果需要在一个进程中(假设为一个 Activity)访问另一个进程中(假设为一个 Service)某个对象的方法，就可以使用 AIDL 来生成接口代码并伪装传递各种参数。

2. AIDL 使用示例

跨进程调用通常是以服务端提供服务供客户端调用的形式存在的。因此要使用 AIDL，服务端需要以 *.aidl 文件的方式提供服务接口，AIDL 工具将自动生成一个对应的 Java 接口对象，并且在生成的接口中包含一个供客户端调用的 Stub 服务桩类，Stub 对象就是远程对象的本地代理。服务端的实现类需要提供返回 Stub 服务桩类对象的方法。使用时，客户端通过 onBind 方法得到服务端 Stub 服务桩类的对象，之后就可以像使用本地对象一样使用它了。接下来通过示例来介绍 AIDL 的具体使用方法。

本实例一共包括了两个 Android Project，即分别为客户端和服务端，客户端项目为 MultiProcessDemo，服务端项目为 MultiProcessDemo_AIDLServer。为了实现通过 AIDL 方式使用远程服务，首先需要使用 *.aidl 文件编写用于使用远程服务的接口，编写接口的方法是在项目的包构下建立后缀名为.aidl 的文件，ADT 会因此判别文件是用于定义接口的，进而自动生成相应的 Java 代码，*.aidl 文件的编写通常较为简捷，只需要指明包名然后简单地定义接口需要实现的方法即可，本项目的 IMusicControlService.aidl 文件内容如下，文件名称通常加上大写字母 I 作为前缀：

```
package com.android.MultiProcessDemo.remote;
interface IMusicControlService{
        void play();
        void stop();
        void pause();
}
```

ADT 会根据上面定义的 IMusicControlService.aidl 文件生成一个 Java 接口类。生成的接口类中包含了一个 Stub 类，通过继承 Stub 类并实现前面所定义的一系列接口方法，然后在客户端绑定 Service 的时候返回该 Stub 对象即可。代码中采用内部匿名类的方式实例化了一个 Stub 对象，然后通过 onBind()方法将该对象传递给调用服务的 Activity。RemoteMusicService 的代码如下：

```
public class RemoteMusicService extends Service {

    private static final String TAG = "RemoteMusicService";
    private MediaPlayer mediaPlayer;

    @Override
    public IBinder onBind(Intent intent) {
        return binder;                        //在服务被绑定时返回该远程对象
    }

    private final IMusicControlService.Stub binder = new IMusicControlService.Stub() {
        //实现 IMusicControlService 接口的匿名内部类
        @Override
        public void stop() throws RemoteException {
            //停止播放,由于是远程调用该方法,因此抛出 RemoteExcaption 异常
            Log.d(TAG,"stop....");
            if (mediaPlayer != null) {
                mediaPlayer.stop();
                try {
                    mediaPlayer.prepare();
                    mediaPlayer.seekTo(0);
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }

        @Override
        public void play() throws RemoteException {     //播放或继续播放
            Log.d(TAG,"play....");
            if (mediaPlayer == null) {
                mediaPlayer = MediaPlayer.create(RemoteMusicService.this,
                        R.raw.tmp);
                mediaPlayer.setLooping(false);
            }
            if (!mediaPlayer.isPlaying()) {
                mediaPlayer.start();
            }
```

```
            }

            @Override
            public void pause() throws RemoteException {     //暂停播放
                Log.d(TAG,"pause....");

                if (mediaPlayer != null && mediaPlayer.isPlaying()) {
                    mediaPlayer.pause();
                }
            }

        };

        @Override
        public void onDestroy() {
            super.onDestroy();

            Log.d(TAG, "onDestroy");
            if(mediaPlayer != null){
                mediaPlayer.stop();
                mediaPlayer.release();
            }
        }
    }
```

服务需要在 MultiProcessDemo_AIDLServer 项目的 AndroidManifest.xml 文件中进行注册，否则在运行时会提示找不到服务，MultiProcessDemo 项目中就不用声明这个服务了。

要在 MultiProcessDemo 中去使用 Server 提供的服务，首先需要将 IMusicControlService.aidl 接口定义文件 IMusicControlService.aidl 包括其包结构复制到 ServiceDemo 下，ADT 也会在项目中自动生成对应的 Java 接口代码；然后就可以通过类似于绑定本地服务的方式来绑定该远程服务了，如图 6-12 所示，需要注意的是在 ServiceConnection 中获取 IBinder 对象时需要通过 IMusicControlService.Stub.asInterface（IBinder）的方式来进行类型转换，而不能够使用前面的直接强制类型转换。PlayRemoteMusic 的关键代码如下：

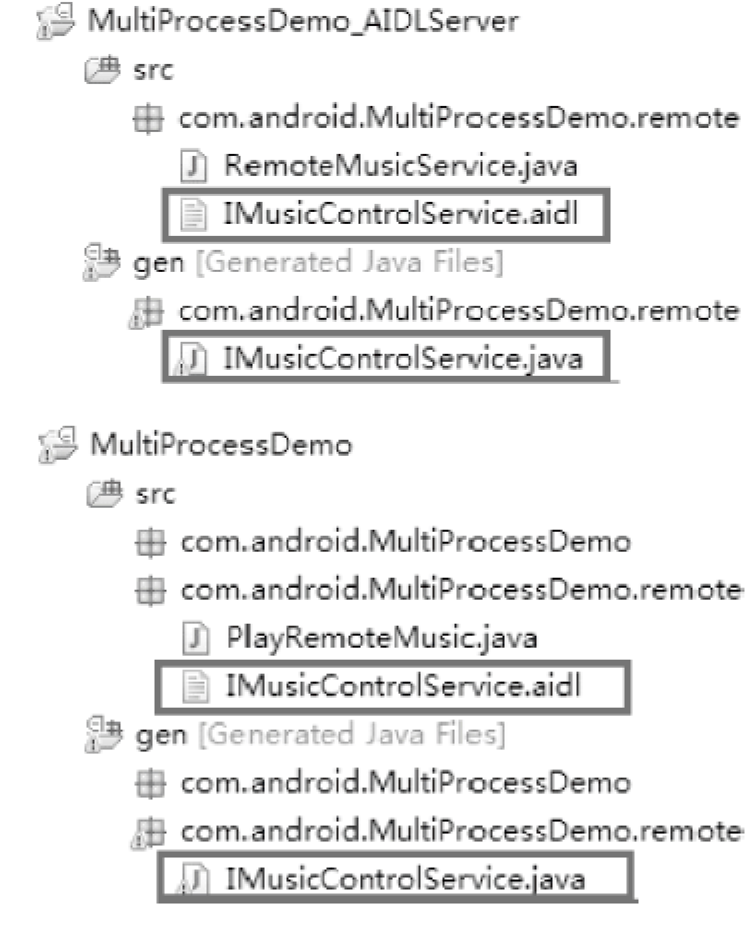


图 6-12 服务端和客户端代码的关系

```
public class PlayRemoteMusic extends Activity implements OnClickListener {

  private IMusicControlService musicService;

  @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.remote_music_service);
        connection();
    }

    private void connection(){                          //绑定远程服务
        Intent intent = new
Intent("com.android.MultiProcessDemo.remote.RemoteMusicService");
         bindService(intent, sc, Context.BIND_AUTO_CREATE);
    }

    @Override
    public void onClick(View v) {
        try{
            switch (v.getId()) {
            case R.id.play://向本地服务一样使用远程服务的方法
                musicService.play();
                break;
            case R.id.stop:
                if(musicService != null){
                   musicService.stop();
                }
              break;
            case R.id.pause:
              if(musicService != null){
                musicService.pause();
              }
              break;
            case R.id.exit:
                this.finish();
                break;
            }
        }catch (RemoteException e) {
            e.printStackTrace();
        }
    }
  private ServiceConnection sc = new ServiceConnection() {

      @Override
      public void onServiceDisconnected(ComponentName name) {
          musicService = null;
      }

      @Override
      public void onServiceConnected(ComponentName name, IBinder service) {
          //获取远程服务对象在本地的代理,必须是通过如下方式进行类型转换
          musicService = IMusicControlService.Stub.asInterface(service);
      }
    };
  }
```

当 Activity 绑定到远程 Service 对象时，onServiceConnected 方法将被调用并获得 IBinder 对象，该对象就是远程 Service 对象在本地的代理。然后借助 AIDL 所定义的接口，

就可以像使用本地服务一样来使用远程服务的方法了。

在示例中的 MultiProcessDemo 下还实现了通过 intent 消息来控制音乐播放服务，原理是 Service 的 onStart()方法可以被反复调用，因此采用在 intent 对象中携带操作码 op 的方式，在 Service 的 onStart()方法中接收 op 并调用对应方法来实现对 Service 的控制，可以结合代码来比较一下两种方式的不同。

6.5 生产者/消费者模型

6.5.1 生产者/消费者模型简介

生产者/消费者问题是一个经典的并发程序同步问题，该问题最早由 Dijkstra 提出，用于演示他提出的信号量机制。这个模型的大致描述为：在同一个进程地址空间内执行的两个线程，生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻塞，直到新的物品被生产出来。

由于模型中的生产者和消费者共享了用于存放产品的缓冲区，就必然会涉及对缓冲访问的互斥问题，如果不能正确处理这种互斥问题，就可能会出现各个线程所访问到的缓冲区不一致的情形，进而导致错误的访问操作。例如，前一个线程正在对共享区域进行访问和更改时，后一个线程也同时对该区域进行访问，那么对于后一个线程来说，前一个线程所进行的更改是不可见的，并最终会导致共享区域的内容的不可预知。

6.5.2 Java 下解决互斥问题

在 Java 中，要跨线程维护正确的可见性，就需要使用 synchronized(或 volatile)关键字以确保一个线程可以看见另一个线程做的更改。为了在线程之间进行可靠的通信，也为了互斥访问，同步是必需的，否则会造成数据不一致(线程安全)。这归因于 Java 语言规范的内存模型，它规定一个线程所做的变化何时以及如何变成对其他线程可见。

Java 以提供关键字 synchronized 的形式，为防止资源冲突提供了内置支持。当任务要执行被 synchronized 关键字保护的代码片段时，它将首先检查锁是否可用，然后再获取锁，执行代码，完成后释放锁。共享资源一般是以对象形式存在的内存片段，但也可以是文件、输入/输出端口或者是打印机等。要控制对共享资源的访问，需要首先将其包装进一个对象，然后把所有要访问这个资源的方法标记为 synchronized。如果某个任务处于一个对标记为 synchronized 的方法的调用中，那么在这个线程从该方法返回之前，其他所有要调用类中任何标记为 synchronized 方法的线程都会被阻塞。

一个任务可以多次获得对象的锁。如果一个方法在同一对象上调用了第二个方法，后者又调用了同一对象上的另一个方法，就会发生这种情况。JVM 负责跟踪对象被加锁的次数。如果一个对象被解锁(即锁被完全释放)，其计数变为 0。在任务第一次给对象加锁时，计数变为 1。每当这个相同的任务在这个对象上获得锁时，计数都会递增。显然，只有首先

获得了锁的任务才能允许继续获取多个锁。每当任务离开一个 synchronized 方法，计数递减，当计数为 0 时，锁被完全释放，此时别的任务就可以使用此资源。

1. synchronized 的用法

首先进行一个约定：假设 P1、P2 是同一个类的不同对象，这个类中定义了以下几种用法情况的同步块或同步方法，P1、P2 都能够调用它们。

（1）把 synchronized 当作函数修饰符时，示例代码如下：

```
public synchronized void method(){
    //…
}
```

这就是同步方法，这时 synchronized 锁定的是调用这个同步方法对象。即当 P1 在不同的线程中执行这个同步方法时，它们之间会形成互斥，达到同步的效果。但是这个对象所属的类所产生的另一对象 P2 却能够任意调用这个被加了 synchronized 关键字的方法。即 P1、P2 互不相关。上边的示例代码等同于如下代码：

```
public void method() {
    synchronized (this)
    {
        //…
    }
}
```

其中的 this 指的就是调用这个方法的对象，如 P1。

（2）同步块，示例代码如下：

```
public void method(Object o) {
    synchronized(o)
    {
        //…
    }
}
```

此处锁就是 o 这个对象，谁拿到这个锁谁就能够运行它所控制的那段代码。当有一个明确的对象作为锁时，就直接使用这个对象；但当没有明确的对象作为锁，只是想让一段代码同步时，可以创建一个特别的 instance（非 static）变量（对象）来充当锁，如下面代码所示：

```
class Foo implements Runnable
{
    // 特别的 instance 变量
    private byte[] lock = new byte[0];
    public void method() {
        synchronized(lock) { //… }
    }
    //…
}
```

注：零长度的 byte 数组对象创建起来将比任何对象都经济——查看编译后的字节码——生成零长度的 byte[ ]对象只需 3 条操作码，而 Object lock = new Object()则需要 7 条操作码。

(3) 将 synchronized 作用于 static 函数，示例代码如下：

```
class Foo{
    public synchronized static void method1()        //同步的 static 函数
    {
        //…
    }
    public void method2()
    {
        synchronized(Foo.class){}                     //class literal(类名称字面常量)
    }
}
```

上面代码中的 method2()方法是把 class literal 作为锁的情况，这与同步的 static 函数产生的效果相同，这种方式取得的锁很特别，是当前调用这个方法的对象所属的类(Class，而不再是由这个 Class 产生的某个具体对象)。

假如一个类中定义了一个 synchronized 的 static 函数 A，也定义了一个 synchronized 的 instance 函数 B，那么这个类的同一对象 Obj 在多线程中分别访问 A 和 B 两个方法时，不会构成同步，因为它们的锁不相同。A 方法的锁是 Obj 所属的那个 Class，而 B 的锁是 Obj 所属的这个对象。

6.5.3 Android 下的示例 Project

本节将介绍一个实现在 Android 上的生产者消费者模型的示例项目，该示例项目名称为 ProducerConsumer，该示例还结合了消息机制来对界面进行更新。该示例工作的示意图如图 6-13 所示。

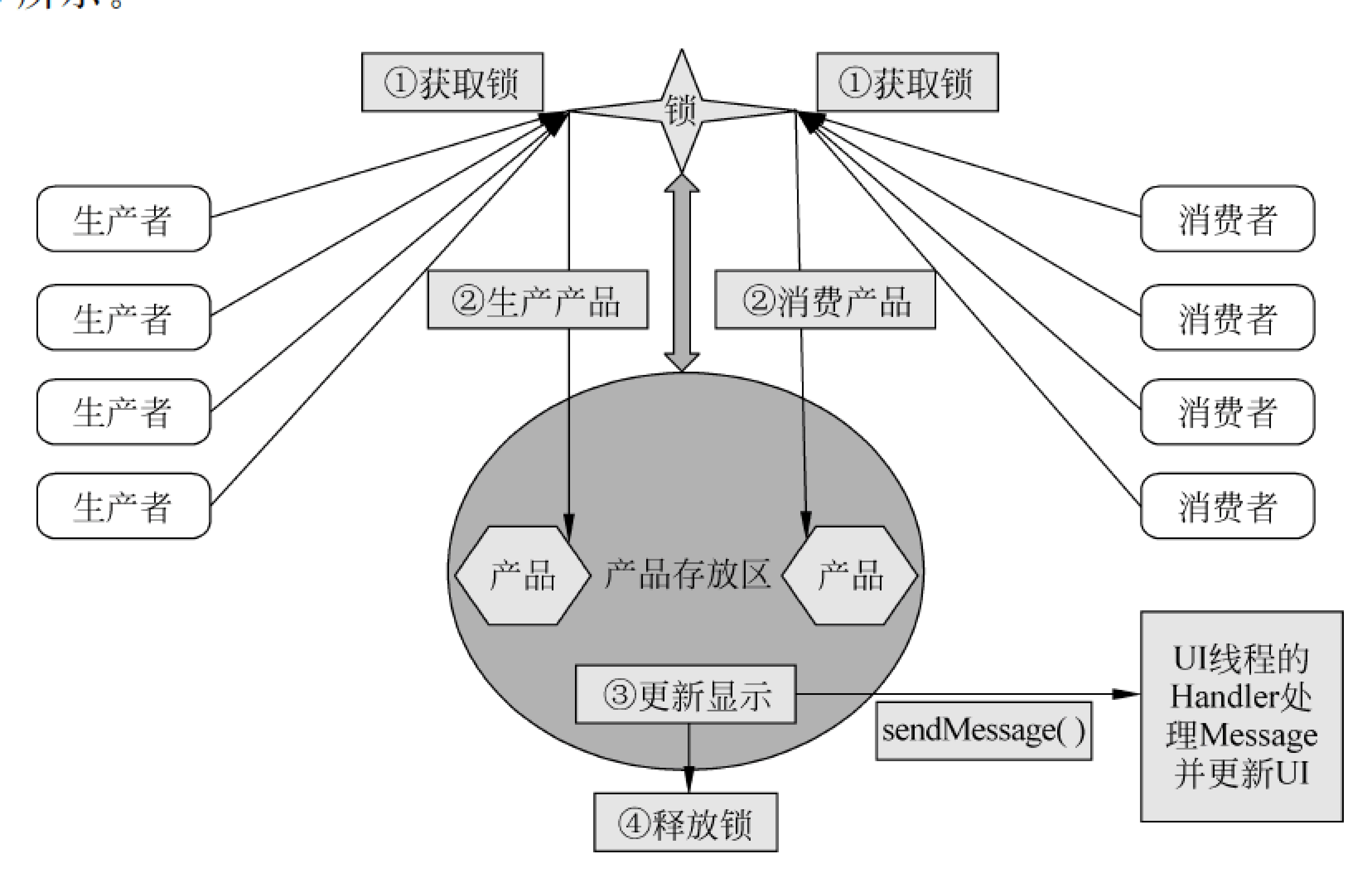


图 6-13 ProducerConsumer 示例工作示意图

1. 生产者

代表生产者的类 Producer.java 的代码如下(由于在代码中添加了比较详细的注释,因此就不再在正文中重复说明,请读者结合注释来理解代码):

```
public class Producer implements Runnable {
    private Handler mHandler;                          //用于向 UI 线程发送消息的 Handler 类
    private static int count;                          //产品计数器,递增的给每一件产品生成
                                                       //唯一的编号
    private static int producerCount;                  //生产者计数器,递增的为每一个生产者
                                                       //线程生成编号
    private static boolean ispause = true;             //模型是否处于暂停状态
    private static boolean isProvided = false;         //该生产者线程是否刚生产过一次产品
    private static int contents_capacity = 3;          //用于存放产品的缓冲区容量
    private List<String> contents;                     //存放产品的容器
    private int number;                                //生产者线程编号

   /**
    * 生产者线程构造方法
    * @param contents 传入的产品存放缓冲区
    * @param mHandler 传入的主线程 Handler
    */
    public Producer(List<String> contents, Handler mHandler) {
        this.contents = contents;
        this.mHandler = mHandler;
        ++producerCount;
        this.number = producerCount;                   //获取自身的线程编号
    }

    @Override
    public void run() {
        while (true) {
        if(!getIspause()){                             //判断线程是否处于暂停状态
            //使用 synchronized 关键字控制多个线程互斥地进入共享区域,
            //若线程获得锁则执行括号内代码,否则线程阻塞
            synchronized (contents) {
                if(contents.size() < getContents_capacity()){
                  String s = "产品" + (count++);         //初始化字符,代表一次生产过程
                  this.contents.add(s);                //将产品加入缓冲区
              String print_log = "生产者" + number + "号生产了" + s;     //日志信息
              //System.out.println(print_log);
              Message message = Message.obtain();      //从消息池中获取一个消息对象
              message.what = 1;                        //1 代表生产者线程
              message.arg1 = contents.size();          //arg1 字段存储当前产品数量
              message.obj = print_log + "\n";          //obj 字段存储日志信息
              mHandler.sendMessage(message);           //发送消息给主线程处理
              this.contents.notifyAll();               //唤醒其他正在等待缓冲区解锁的线程
              isProvided = true;                       //将本线程标记为刚生产过一次产品
            }
          }
        }
```

```
        //若 isProvided 字段为真,则表示本线程刚生产过一次产品,
        //额外睡眠一段时间用于模拟生产一个产品所需消耗的时间
        if( isProvided){
          try {
            Thread.sleep(1000 * 2);
            //睡眠一段时间后置 isProvided 字段为 false,表示可继续生产产品
            isProvided = false;
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
        try {
          Thread.sleep(500);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }

    //以下为 getter 和 setter 方法
    public static boolean getIspause() {
     return ispause;
    }

    public static void setIspause(boolean ispause) {
     Producer.ispause = ispause;
    }

    public static int getContents_capacity() {
     return contents_capacity;
    }

    public static void setContents_capacity(int contents_capacity) {
    Producer.contents_capacity = contents_capacity;
    }
}
```

2. 消费者

代表消费者的类 Consumer.java 的代码如下：

```
public class Consumer implements Runnable {
    private Handler mHandler;              //用于向 UI 线程发送消息的 Handler 类
    private static int consumerCount;      //消费者计数器,递增的为每一个消费者线程生
                                           //成编号
    private static boolean ispause = true;   //模型是否处于暂停状态
    private List<String> contents;         //存放产品的容器
    private int number;                    //消费者线程编号
    private boolean isServed = false;      //该消费者线程是否刚消费过一次产品

    /**
```

```
     * 消费者线程构造方法
     * @param contents 传入的产品存放缓冲区
     * @param mHandler 传入的主线程 Handler
     */
    public Consumer(List<String> contents, Handler mHandler) {
      this.contents = contents;
      this.mHandler = mHandler;
      ++consumerCount;
      number = consumerCount;    //获取自身的线程编号
    }

    @Override
    public void run() {
      while (true) {
        //判断线程是否处于暂停状态,若 ispause == false 则执行括号内代码
        if(!getIspause()){
          //使用 synchronized 关键字控制多个线程互斥地进入共享区域,
          //若线程获得锁则执行括号内代码,否则线程阻塞
          synchronized (contents) {
            //判断产品缓冲区是否已空,若已为空则执行
            //wait()方法释放锁并使当前线程进入等待队列
            if (contents.isEmpty()) {
              try {
                contents.wait();
              } catch (InterruptedException e) {
                e.printStackTrace();
              }
            } else {     //若产品缓冲区不为空,则取出产品进行"消费"
              String print_log = "消费者" + number + "号消费了"
                  + contents.remove(0);    //取出并消费产品,生成日志信息
              //System.out.println(print_log);
              Message message = Message.obtain();  //从消息池中获取一个消息对象
              message.what = 2;                    //2 代表为消费者线程
              message.arg1 = contents.size();      //arg1 字段存储当前产品数量
              message.obj = print_log + "\n";      //obj 字段存储日志信息
              mHandler.sendMessage(message);       //发送消息给主线程处理
              isServed = true;                     //将本线程标记为刚消费过一次产品
                                                   //的状态
            }
          }
          //若 isServed 字段为真,则表示本线程刚消费过一次产品,
          //额外睡眠一段时间用于模拟消费一个产品所需消耗的时间
          if(isServed){
            try {
              Thread.sleep(1000 * 3);
              //睡眠一段时间后置 isServed 字段为 false,表示可继续消费产品
              isServed = false;
            } catch (InterruptedException e){
              e.printStackTrace();
```

```
                }
            }
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

//以下为 getter 和 setter 方法
public static boolean getIspause() {
    return ispause;
}

public static void setIspause(boolean ispause) {
    Consumer.ispause = ispause;
}
}
```

3. 模型演示

为了方便生产者/消费者模型的演示，这里实现了一个内容较为丰富的用户界面，包括了显示当前的生产者数量、消费者数量、仓库容量以及操作模型运行或暂停的两个按钮，还有一个用于实时显示当前产品数量的文本，此外还设置了两个用于输出生产和消费日志的文本框，界面的实现相对简单，代码就不再列出，这里仅给出用于处理日志输出以及实时更新产品数的 MessageHandler 代码：

```
class MessageHandler extends Handler {
    public MessageHandler() {
        super();
    }

    @Override
    public void handleMessage(Message msg) {
        switch(msg.what){
        case 1: //消息来自于生产者线程
            producer_log.append((String)msg.obj);
            num_products.setText(msg.arg1 + "个");
            break;
        case 2: //消息来自于消费者线程
            consumer_log.append((String)msg.obj);
            num_products.setText(msg.arg1 + "个");
            break;
        }
    }
}
```

示例的运行效果如图 6-14 所示。

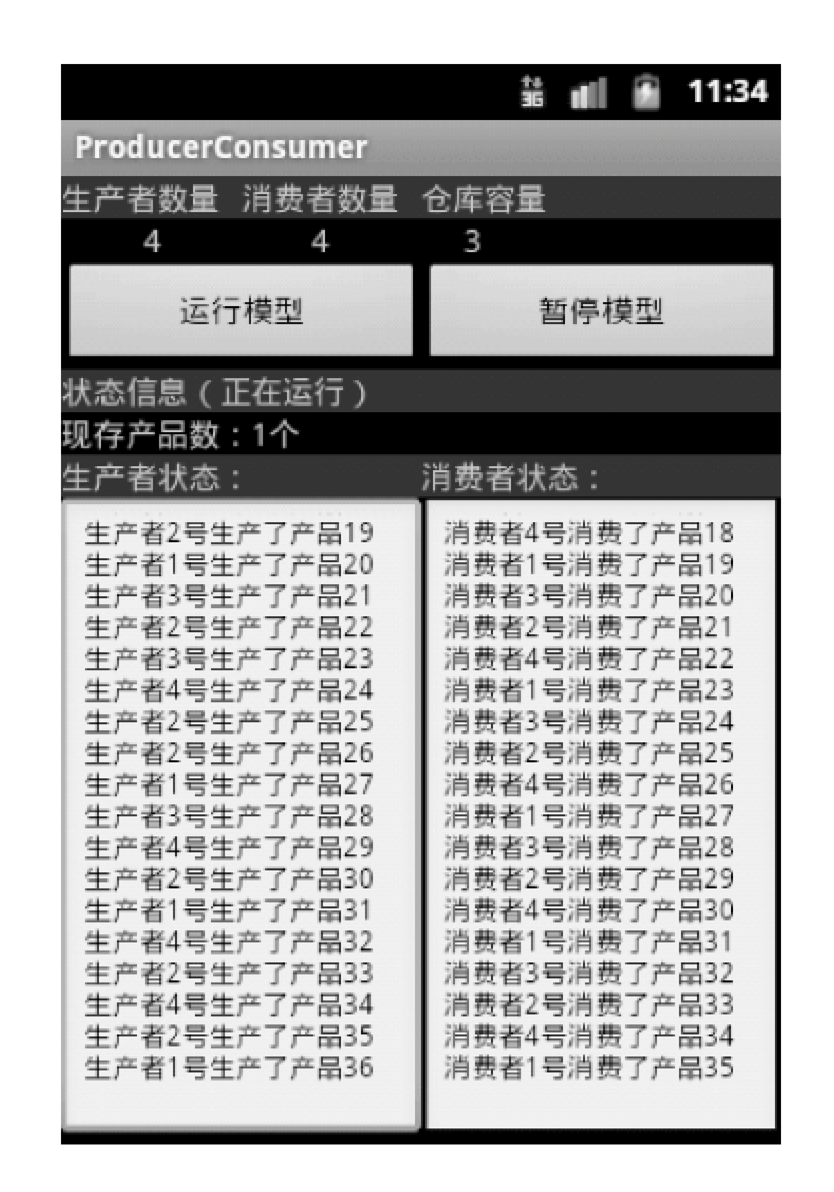


图 6-14　生产者消费者模型实现效果

参考文献

1. 进程与线程简介：http://www.sf.org.cn/blog/symbianapi/symbianossdk/symbianosguide/base/usinguserlibrary/memorymanagement/concepts/200605/18310.html.
2. Android 核心分析 之九——Zygote Service：http://blog.csdn.net/maxleng/article/details/5508488.
3. 多线程和多进程的区别：http://blog.csdn.net/hairetz/article/details/4281931.
4. Android 进程与线程：http://www.cnblogs.com/feisky/archive/2010/01/01/1637409.html.
5. Android 应用程序模型(应用程序、任务、进程、线程)：http://blog.csdn.net/iefreer/article/details/4460196.
6. Android 进程间通信——消息机制及 IPC 机制实现：http://myqdroid.blog.51cto.com/2057579/394189.
7. Android 线程中通信：http://blog.csdn.net/familygo/article/details/6693007.
8. 深入剖析 Android 消息机制：http://blog.csdn.net/coolszy/article/details/6360577.
9. Android 进程间通信(使用 AIDL)：http://blog.csdn.net/saintswordsman/article/details/5130947.
10. 维基百科“Producer-consumer problem”：http://en.wikipedia.org/wiki/Producer-consumer_problem.

第7章 多媒体编程

多媒体(Multimedia)的含义是：在计算机系统中组合两种或两种以上媒体的一种人机交互式的信息交流方式，这些媒体包括文字、图片、照片、声音、动画和影片等。从多媒体的概念提出到今天，它的应用领域已经涉及诸如广告、艺术、教育、娱乐、工程、医药、商业及科学研究等行业，尤其是网络技术日益发达的今天，多媒体在网页上也得到了广泛的应用，正因为有了多媒体的应用，使得应用程序从以前单调乏味的形式发展到今天这样丰富多彩的形式。本章就是来介绍如何在 Android 上应用多媒体，达到让应用程序美观、易用并且具有吸引力的效果。随着 Android 版本的快速更新，其对多媒体的支持水平也在快速提升，本章主要介绍的内容有音频、视频、绘图及 OpenGL 等，由于图片的显示相对简单，直接嵌入 ImageView 即可，在前面也已经广泛接触到，本章就不再花篇幅进行介绍了。

7.1 音视频支持

在多媒体应用中，最常用的就是音频和视频功能了。例如，一款设计良好的游戏必定配合了与其各种场景相适应的音效、背景音乐等，这些音频效果将可以给玩家强烈的游戏带入感。音效可以增添游戏的趣味性、真实性以及强烈的操作快感；背景音乐则可以为游戏情节的发展提供支持。对于非游戏类的其他应用，音频也广泛地作为操作提醒提示音而被使用。可以说，一款没有音频元素的应用是不能够充分地优化用户体验的，当然对于音频播放器来说，音频播放更是其业务实现的核心。而对于视频，除了提供本地播放、在线播放视频等功能外，也可以广泛应用到各种应用程序中，例如作为游戏的开场/过场动画而起到对剧情的描述、推动作用；作为应用程序的使用介绍等。总之，音视频支持是应用程序中十分重要的一个部分，因此，本节将开始介绍如何在 Android 平台下开发具有音视频功能的应用程序。

7.1.1 播放音频

播放音频涉及对音频文件的解码，目前 Android 目前对一些主流的音频格式都有着较好的支持，原生 Android 系统所支持解码即播放的音频格有 AAC、AMR、WAV、MP3、WMA、OGG、MIDI 及 FLAC(3.1＋)等(对于 Android 模拟器来说，暂时只支持 OGG、WAV 和 MP3 3 种格式)。对于通常的应用来说已经远远足够，如果需要播放某些特殊格式的音频文件，则需要实现用于解码该格式音频文件的解码器，这部分内容已经超出了本书的

讨论范围,有兴趣的读者可以通过其他书籍进行了解。

在开发应用程序时,如果需要在应用程序中播放指定的音频文件,通常将需要使用的音频文件存放在 res/raw 文件夹下,ADT 会将这个资源在 R.java 中进行关联。借助于 Android 提供的 MediaPlayer 类可以快速地完成播放一段音频的代码实现,创建一个 MediaPlayer 对象即可,创建该类对象的方式有如下两种方法:

- 一是可以使用静态方法 MediaPlayer.create()创建,通过参数使播放器与资源相关联起来,再使用 start()方法开始播放指定的音频文件。
- 二是使用构造方法 MediaPlayer()创建一个播放器对象,然后使用播放器的 setDataSource()方法将音频资源相关联,与静态方法创建播放器不同的是,使用构造方法创建的播放器还需要首先使用 prepare()方法,然后再使用 start()方法开始播放;否则会抛出一个播放器状态不正常的异常。

使用静态方法创建播放器对象并播放音频的代码如下:

```
MediaPlayer mediaPlayer = MediaPlayer.create(this, R.raw.tmp);
mediaPlayer.start();
```

使用构造方法创建播放器对象并播放音频的代码如下:

```
MediaPlayer mp = new MediaPlayer();
mp.setDataSource(PATH_TO_FILE);
mp.prepare();
mp.start();
```

通常对音频的播放会通过 Service 组件的方式来实现,这样可以使得音频播放的进程不依赖于某个 Activity,从而达到在 Activity 关闭时音频在后台播放的效果,下面就实现一个在 Service 中播放音频的示例。

1. 实现音频播放 Service

实现音频播放的 Service 代码如下:

```
public class MusicService extends Service {

    private static final String TAG = "MyService";
    private MediaPlayer mediaPlayer;

    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

    @Override
    public void onCreate() {
        Log.v(TAG, "onCreate");
        Toast.makeText(this, "启动音乐播放器", Toast.LENGTH_SHORT).show();
        //若当前 mediaPlayer 对象为空,则创建一个媒体播放器对象用于播放音频文件
        if (mediaPlayer == null) {
            //使用指定的音频文件初始化播放器
```

```
                mediaPlayer = MediaPlayer.create(this, R.raw.tmp);
                mediaPlayer.setLooping(false);  //设置为不循环播放
            }
        }

        @Override
        public void onDestroy() {
            Log.v(TAG, "onDestroy");
            Toast.makeText(this, "停止音乐播放器", Toast.LENGTH_SHORT).show();
            if (mediaPlayer != null) {      //当 Service 被销毁时释放 mediaPlayer 所占资源
                mediaPlayer.stop();
                mediaPlayer.release();
            }
        }

        @Override
        public void onStart(Intent intent, int startId) {
            Log.v(TAG, "onStart");
            if (intent != null) {
                Bundle bundle = intent.getExtras();
                if (bundle != null) {
                    int op = bundle.getInt("op");//取出绑定在 intent 上的操作数值
                    switch (op) {               //根据不同的操作数执行对应的操作
                    case PlayMusic.OP_PLAY:
                        play();
                        break;
                    case PlayMusic.OP_PAUSE:
                        pause();
                        break;
                    case PlayMusic.OP_STOP:
                        stop();
                        break;
                    default:
                        break;
                    }

              }
          }

        }

        public void play() {                    //播放音乐
            if (!mediaPlayer.isPlaying()) {
                mediaPlayer.start();
            }
        }

        public void pause() {                   //暂停,可以继续播放
            if (mediaPlayer != null && mediaPlayer.isPlaying()) {
                mediaPlayer.pause();
            }
        }

```

```
70      public void stop() {                    //停止,不保留播放位移,下次从头播放
71          if (mediaPlayer != null) {
72              mediaPlayer.pause();
73              mediaPlayer.seekTo(0);
74          }
75      }
76  }
```

如上面的代码所示，在 Service 中实现了对音频播放器的控制功能接口，包括：

- play()——播放音频，代码第 58～62 行。
- pause()——暂停音频播放，代码第 64～68 行。
- stop()——停止音频播放，代码第 70～75 行。

在 Service 的 onCreate()方法中包含了创建音频播放器对象的代码(第 15～20 行)；onDestroy()方法中包含了销毁音频播放器对象的代码(第 27～30 行)；onStart()方法中包含了对从 Activity(用于控制音频播放的界面)发送过来的消息的处理的代码(第 40～51 行)。在本例中，Service 的生命周期为：onCreate()→onStart()(多次)→onDestroy()。

2. 实现音频播放控制 Activity

在本节第 1 部分中已经实现了用于播放音频的 Service，Service 为控制音频提供了 3 个接口，通过 Service 的代码可以知道，对音频的控制是通过在调用 startService()方法时传递一个整型的控制码(op)的方式来进行的，为此在 Activity 中定义了 4 个全局常量：

```
public static final int BAD_REQUEST = -1;
public static final int OP_PLAY = 1;
public static final int OP_PAUSE = 2;
public static final int OP_STOP = 3;
```

这 4 个常量分别代表了一种对音频播放的控制操作，当需要对音频播放执行播放、暂停和停止操作时，只需要通过 startService(intent)方法调用 Service 并通过 intent 传递相应的操作码即可：

```
        int op = BAD_REQUEST;
        //用于启动对应 Service 的 intent 对象
        Intent intent = new Intent("com.android.ServiceDemo.musicService");
        switch (v.getId()) {            //根据被点击按钮生成操作码 op,进行相应的操作
            case R.id.play:
                op = OP_PLAY;
                break;
            case R.id.pause:
                op = OP_PAUSE;
                break;
            case R.id.stop:
                op = OP_STOP;
                break;
            case R.id.close:
                this.finish();
            break;
```

```
17            case R.id.exit:
18                stopService(intent);
19                    this.finish();
20                    return;
21            }
22            Bundle bundle = new Bundle();
23            bundle.putInt("op", op);              //将 op 存放于 bundle 中
24            intent.putExtras(bundle);             //将 bundle 绑定到 intent
25            startService(intent);                 //使用 intent 启动服务
```

上面的代码是位于 Activity 的 onClick(View v)方法中的，即当 Activity 上的视图控件发生被点击的事件时将会触发此方法，当某个按钮(播放、暂停或停止)被单击时，该按钮对象将会被传入，通过 v.getId()方法获取被单击的具体按钮，然后以此为依据对 op 进行赋值(第 05、06、08、09、11、12 行)，然后将 op 存放于一个 Bundle 对象中(第 22 行和第 23 行)，最终将 Bundle 对象绑定到 intent 并调用 startService()方法来实现对 Service 音频播放的控制(第 24 行和第 25 行)。

7.1.2 录制音频

在 7.1.1 节中已经介绍了如何播放音频文件，本节将介绍它的逆过程，即如何在 Android 平台上录制音频，尽管该功能并不会被大多数应用所使用，但是对录制音频的实现有一个简单的了解也是有好处的。目前 Android 支持编码的音频格式有 AAC 和 AMR。录制音频资源最方便的方式即是使用 MediaRecorder 类，通过设置录制音频来源(通常是设备默认的麦克风)就能方便地录制语音，具体包括如下一些步骤：

- new 一个 android.media.MediaRecorder 实例。
- 使用 MediaRecorder.setAudioSource()方法来设置音频资源；这将会很可能使用到 MediaRecorder.AudioSource.MIC。
- 使用 MediaRecorder.setOutputFormat()方法设置输出文件格式。
- 用 MediaRecorder.setAudioEncoder()方法来设置音频编码。
- 使用 setOutputFile()方法设置输出的音频文件。
- 最后，使用 prepare()和 start()方法开始录制音频，通过 stop()和 release()方法完成一段音频的录制。

下面给出一段完整的代码：

```
01    recorder = new MediaRecorder();
02    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
03    recorder.setOutputFormat(MediaRecorder.OutputFormat.DEFAULT);
04    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
05    recorder.setOutputFile(mRecAudioFile.getAbsolutePath());
06    recorder.prepare();
07    recorder.start();
08    recorder.stop();
09    recorder.release();
```

其中，第 01 行实例化了一个 MediaRecorder 对象；第 02 行设置了音频输入的来源(麦

克风)；第 03 行设置了文件输出的格式；第 04 行设置了编码音频的方式；第 05 行设置了输出录制文件的路径；第 06 行和第 07 行则是开始录音；第 08 行和第 09 行在录音完成后调用。

另外，由于录制音频需要使用麦克风，因此需要在 AndroidManifest. xml 声明使用麦克风的权限：

```
<uses-permission android:name="android.permission.RECORD_AUDIO"></uses-permission>
```

7.1.3 播放视频

Android 原生系统所支持解码的视频编码格式有：H. 263(后缀. 3gp 和. mp4)、H. 264 AVC(3. 0+版本，后缀为. 3gp 和. mp4 等)、MPEG-4 SP(后缀为. 3gp)和 VP8(2. 3. 3+版本，后缀为. webm)，其中 H. 263 和 H. 264 是 Android 支持的编码，有了前面学习的音频播放和录制作为基础，再来学习对视频的播放及录制就比较容易了，它们的实现方式非常相似，不同点是音频本身并不会表现为用户界面，而视频则需要成为用户界面的一部分，因此视频播放要使用 VideoView 类来实现，而视频录制也是借助于 MediaRecorder 类，不同之处是此处的数据来源由麦克风变为摄像头，另外再将输出格式及编码方式做相应修改即可，当然，为了实现有实用性的视频录制功能，还需要增加一个用于显示实时录制图像的视图(即视频捕获预览)。

使用 VideoView 播放视频的代码如下，还包括了使用系统提供的播放控制器 MediaController 与 VideoView 进行绑定，从而控制视频的播放/暂停、快进/快退的简单操作。

```
01  Context context = getApplicationContext();
02  VideoView mVideoView = new VideoView(context);
03  mVideoView = (VideoView) findViewById(R.id.surface_view);
04  mVideoView.setVideoURI(Uri.parse(path));
05  MediaController mc = new MediaController(this);
06  mc.setAnchorView(mVideoView);
07  mVideoView.setMediaController(mc);
08  mVideoView.requestFocus();
09  mVideoView.start();
```

其中，第 01～03 行代码用于新建一个 VideoView 对象并绑定到视图；第 04 行代码用于设置需要播放的视频文件 Uri；第 05～07 行代码用于为 VideoView 绑定播放控制器；第 08 行和第 09 行代码使视频播放视图获得焦点并开始播放。

7.1.4 录制视频

如前所述，录制视频的基本实现与录制音频的方式相似，但是由于需要提供一个实时观察录制区域的显示(视频捕获预览)，因此本节将主要对这一部分的实现进行介绍。

为了实现对摄像头捕获图像的预览，需要实现一个 Preview 类，该类继承自 SurfaceView 并实现了 SurfaceHolder. Callback 接口。SurfaceView 可以理解为可嵌入到界面布局中的

一块用于图像绘制的区域，这种 View 通常用于摄像头预览、游戏界面、3D 绘图等，前面用于播放视频的 VideoView 就是 SurfaceView 的一个子类。每一个 SurfaceView 都有一个与之绑定的 SurfaceHolder 类对象用于控制 SurfaceView 的一些属性，可以通过 SurfaceView 的 getHolder()方法获取到 SurfaceHolder 实例，SurfaceHolder. Callback 接口则提供了用于在 SurfaceView 创建、更改和被销毁时所调用的回调方法，通过实现这个接口来进行 SurfaceView 的初始化，更新及销毁的工作。

Preview 类主要就是实现了 SurfaceHolder. Callback 接口的 3 个回调方法，并且加入了用于监听开始录制和停止录制的按键事件方法，以及用于控制视频开始录制和停止录制的方法。3 个回调方法分别为：

- surfaceChanged()——当视图发生结构上的变化(如尺寸)时被调用，在本例中该方法为空。
- surfaceCreated()——当视图创建时被调用，包含初始化录像功能的代码。
- surfaceDestroyed()——当视图销毁时被调用，主要用于释放相关的资源。

surfaceCreated()方法的代码如下：

```
01  if(mMediaRecorder == null){
02      mRecVideoFile = new File("/mnt/sdcard/VideoRecordTempFile.3gp");
03      mMediaRecorder = new MediaRecorder();
04      mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
05      mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
06      mMediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
07      mMediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
08      mMediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H263);
09      mMediaRecorder.setOutputFile(mRecVideoFile.getAbsolutePath());
10      mMediaRecorder.setPreviewDisplay(holder.getSurface());
11      try {
12          mMediaRecorder.prepare();
13      } catch (IllegalStateException e) {
14          e.printStackTrace();
15      } catch (IOException e) {
16          e.printStackTrace();
17      }
18  }
```

其中，第 02 行代码指定了需要播放的视频文件；第 04 和第 05 行用于设置视频录制设备以及音频录制设备，即手机的摄像头和麦克风；第 06～09 行则是设置输出视频文件的编码方式及格式；第 10 行为视频录制过程指定了预览视图，其中的 holder 对象在 Preview 类的构造方法中通过如下代码获取：

```
public Preview(Context context) {
    super(context);
    holder = this.getHolder();                               //获取 SurfaceHolder 对象
    holder.addCallback(this);                                //添加回调接口
    holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS); //设置缓冲类型
    holder.setFixedSize(400, 300);//设置视图大小
}
```

surfaceDestroyed ()方法的代码如下，其作用是停止并释放 MediaRecorder 对象：

```
public void surfaceDestroyed(SurfaceHolder holder) {
    Log.i(TAG, "surfaceDestroyed");
    mMediaRecorder.stop();
    mMediaRecorder.release();
}
```

上面的代码主要是配置了 MediaRecorder 对象，此外需要添加的就是实现视频开始录制和结束录制并保存这两个操作，本例使用了 DPAD_CENTER 即方向导航键中键作为录制键，读者可以采用任意可用的按键来替换，相关代码如下：

```
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        switch(keyCode){
               case KeyEvent.KEYCODE_DPAD_CENTER:{ //方向导航键中键
                if(mMediaRecorder != null){
                    if(isRecording){
                        finishRecordVideo();
                        isRecording = false;
                    }
                    else{
                        startRecordVideo();
                        isRecording = true;
                    }
                }
                break;
            }
            case KeyEvent.KEYCODE_BACK:{
                System.exit(0);
            }
        }
        return super.onKeyDown(keyCode, event);
    }

    public void startRecordVideo(){
        try{
            mRecVideoFile = File.createTempFile(strTempFile, ".3gp", mRecVideoPath);
            mMediaRecorder.setOutputFile(mRecVideoFile.getAbsolutePath());
            mMediaRecorder.setPreviewDisplay(holder.getSurface());
            mMediaRecorder.prepare();
            mMediaRecorder.start();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

    public void finishRecordVideo(){
        if (mRecVideoFile != null)
        {
            mMediaRecorder.stop();
```

```
41            mMediaRecorder.release();
42        }
43    }
```

如上面的代码所示，第 03～13 行代码即是用于响应 DPAD_CENTER 键被按下事件的，根据当前是否处于录制的状态来开始/完成视频的录制，开始录制的代码在第 25～29 行，完成录制的代码是第 40 行和第 41 行。

7.2 动画效果

动画效果是应用中的一个重要组成部分，可以理解为多媒体的一种，在应用中使用合适的动画效果可以使得界面更具吸引力，同时也能够丰富界面所能够向用户传达的信息。在早期的 Android 版本中的动画主要是基于 View Animation System（视图动画系统），运用这个动画框架可以实现补间动画和帧动画的效果，但是这个动画系统存在着几点缺陷：它只能够将动画应用到 View（视图）对象上，如果对象并不是 View 类型的，这个动画系统就不能够被应用到对象上；另外，借助这个动画系统所实现的动画在运行时，它仅仅将目标 View 对象在屏幕上的绘制图像进行了修改，而没有真正地将这个 View 对象整体进行修改，这就会导致类似于如下的错误：一个按钮对象在动画效果运行时，它实际上所能够进行按键事件响应的区域与它在屏幕上所绘制到的区域并不一致，这可能会造成奇怪的用户体验。

由于 View Animation System 所存在的缺陷，在 Android 3.0 版本之后又加入了另一个动画系统框架，叫做 Property Animation System（属性动画系统）。在 Property Animation System 中很好地解决了在 View Animation System 上面所存在的问题。运用这个框架可以对任意的 Property（例如颜色、位置、尺寸等）赋予动画效果，因此比较灵活。

当然 View Animation System 并不是就被 Property Animation System 完全取代了，View Animation System 在实现动画效果时只需要较少的代码以及简单的设置工作，并且使用它也能够完成很多类型的动画效果，因此如果 View Animation System 能够很好地实现所需的效果，仍然应该首先考虑使用它来实现。

本节首先介绍由 View Animation System 所实现的两种动画——Frame Animation 和 Tween Animation，然后介绍 Property Animation System 的具体用法，最后再介绍一下比较常用的 GIF 格式图片动画的浏览方法。

7.2.1 帧动画（Frame Animation）

帧动画就是需要为整个动画过程中的每一帧都单独地准备图像，类似于电影院使用胶片来播放电影的方式，通过相对快速的图片切换，利用人眼的视觉残留特点来形成的动画效果。本节配套示例名称为 FrameAnimationDemo。

在 Android 中要实现这种类型的动画，使用 AnimationDrawable 类即可，Android 将帧动画简单地看作是一种 Drawable 类型，要实例化一个 AnimationDrawable，要通过一个单独的 xml 文件，这个 xml 文件应建立在 res/drawable 目录下，内容类似于如下形式：

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <animation-list xmlns:android="http://schemas.android.com/apk/res/android"
03     android:oneshot="false">
04   <item android:duration="50" android:drawable="@drawable/firefox_animation_0" />
05   <item android:duration="50" android:drawable="@drawable/firefox_animation_1" />
06   <item android:duration="50" android:drawable="@drawable/firefox_animation_2" />
07 </animation-list>
```

用于帧动画的 xml 文件的根节点都是<animation-list>，这个根节点包含了一系列的<item>节点，分别表示了动画中的某一帧。第 03 行的 android:oneshot 属性的含义是该动画是否循环播放，当该值为 true 时动画仅播放一次，值为 false 时动画循环播放；每个<item>节点下包含了两个属性：android:duration 表示该帧图片持续的时间（单位是 ms），android:drawable 属性则指定这帧图片的内容。本例中使用的是一系列的 Firefox 浏览器 Logo 图片来实现的一个旋转 Logo 动画，动画一共包含了 25 个不同的帧图片，每一个图片和另一个图片上"狐狸"的角度都不一样，如图 7-1 所示。

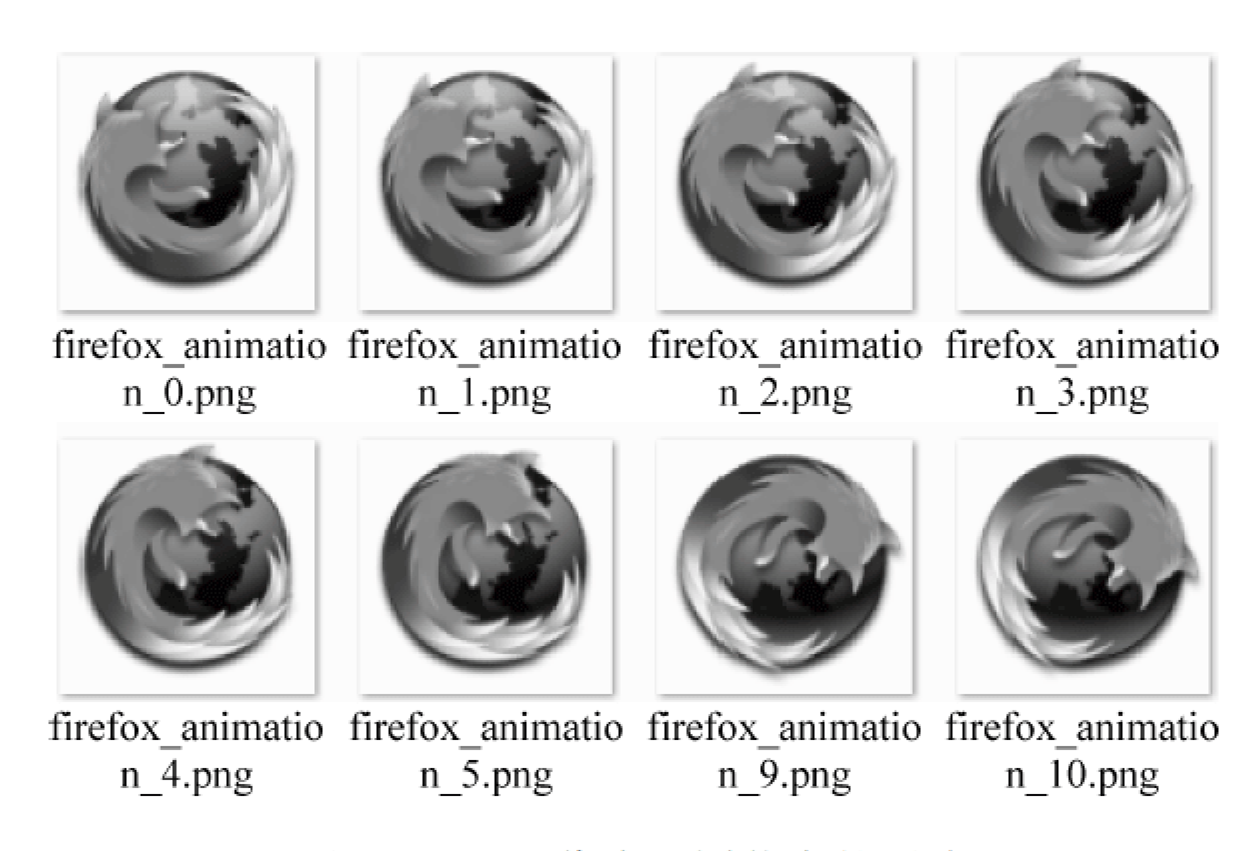


图 7-1　一些代表不同状态的图片

如图 7-1 所示，这些图片通过/res/drawable 下的 firefoxanimation.xml 文件进行组织，然后在 Activity 的 main.xml 布局文件中进行设置，如下面代码的第 9 行所示：

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:layout_height="fill_parent" android:layout_width="fill_parent"
04     android:orientation="vertical">
05     <TextView android:layout_height="wrap_content"
06         android:layout_width="fill_parent" android:text="Android 帧动画示例"
07         android:textSize="20sp" />
08     <ImageView android:layout_height="wrap_content"
09         android:layout_width="wrap_content"
10         android:background="@drawable/firefox_animation"
11         android:id="@+id/AnimationView" android:layout_margin="10dp">
12     </ImageView>
13     <Button android:layout_height="wrap_content" android:text="执行动画"
14         android:textSize="20sp" android:id="@+id/PlayAnimation"
15         android:layout_width="wrap_content">
```

```
    </Button>
    <Button android:layout_height="wrap_content" android:text="暂停动画"
        android:textSize="20sp" android:id="@+id/PauseAnimation"
        android:layout_width="wrap_content">
    </Button>
</LinearLayout>
```

该示例还添加了用于暂停和继续执行动画的按钮,直接调用 AnimationDrawable 的 start()和 stop()方法即可,FrameAnimationActivity 的代码如下:

```
public class FrameAnimationActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button play = (Button) findViewById(R.id.PlayAnimation);
        Button pause = (Button) findViewById(R.id.PauseAnimation);
        final ImageView animationView = (ImageView) findViewById(R.id.AnimationView);

        OnClickListener buttonClickListener = new OnClickListener() {
            @Override
            public void onClick(View v) {

                AnimationDrawable frameAnimation = (AnimationDrawable)
                        animationView.getBackground();

                switch (v.getId()) {
                case R.id.PlayAnimation:
                    frameAnimation.start();
                    break;
                case R.id.PauseAnimation:
                    frameAnimation.stop();
                    break;
                }
            }
        };

        play.setOnClickListener(buttonClickListener);
        pause.setOnClickListener(buttonClickListener);
    }
}
```

示例的运行效果如图 7-2 所示。

7.2.2 补间动画(Tween Animation)

所谓补间动画,简单地说就是不再需要像帧动画那样人工地提供动画效果中的每一帧图像,而是以一个图像作为基础,通过对这个图像进行一系列的诸如缩放、旋转、平移和透明化等变换操作来实现动画效果。

例如,当需要实现 7.2.1 节介绍的旋转效果时,不需要再准备如此多的一组图片,只需

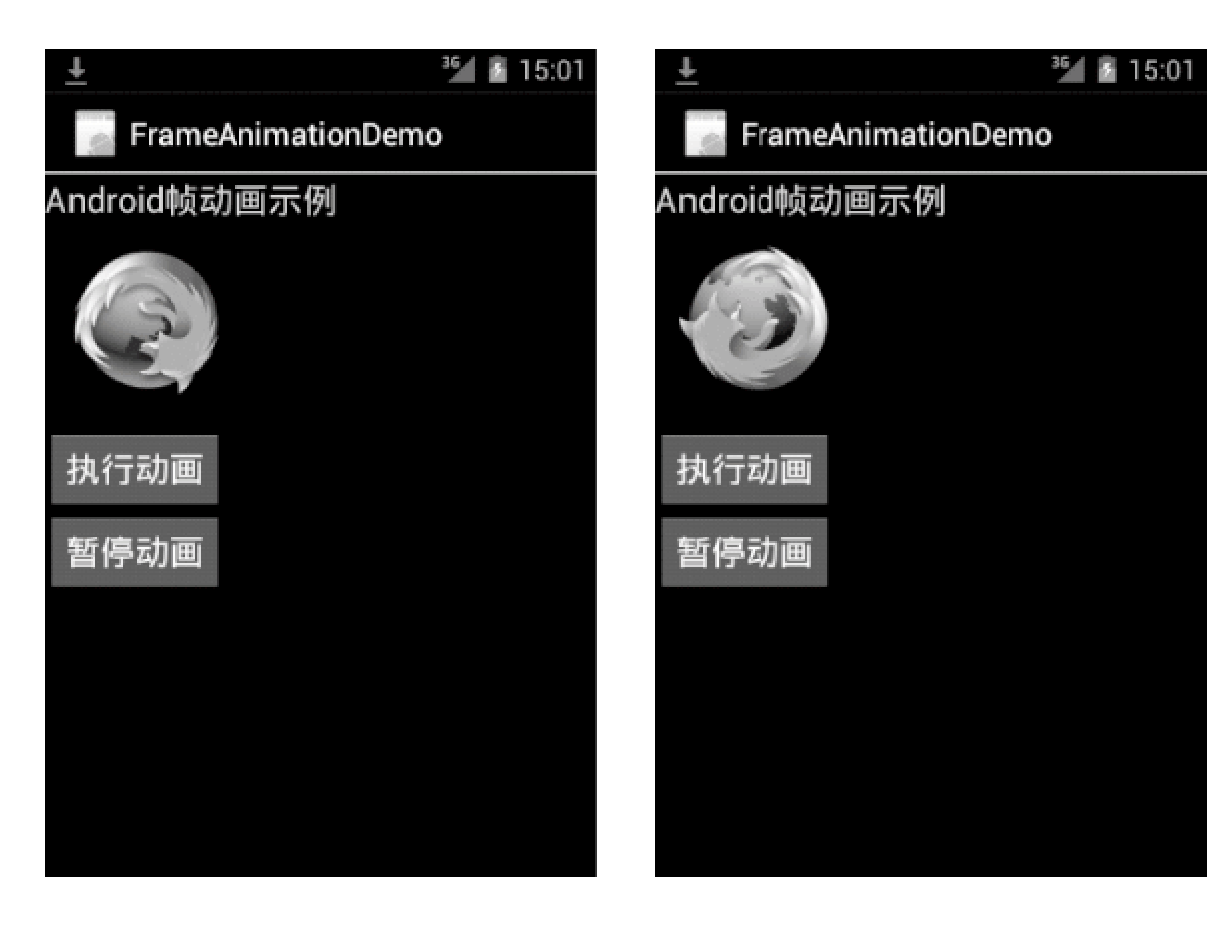


图 7-2　帧动画的两个状态

要一张图片即可(实际上仅使用一张图片并不能达到前面帧动画所实现的效果,而是需要两张,因为 Logo 中的“狐狸”和“地球”是分离的,进行转动的实际上只有“狐狸”,这里说成一张并不影响对补间动画的理解),通过实时的计算就能够得到动画效果的其他状态。

Android 提供的补间动画类型大致包括了如下 4 种(以 xml 标签进行说明):

- <alpha>——透明,即可以动态地调整图像的透明度。
- <rotate>——旋转,对某个对象进行旋转操作。
- <scale>——尺寸,既可以动态地调整图像的尺寸大小,也可以理解为缩放。
- <translate>——平移,对某个对象进行平移的操作。

虽然只提供了这 4 种动画,但是通过混合这几种效果就可以得到十分丰富的动画效果,本节的配套示例项目名称为 TweenAnimation,下面通过对这个示例的说明来进一步了解补间动画的应用方法。

补间动画的信息需要以 xml 文件的形式建立在项目的 res/anim 目录下,本示例一共实现了 5 种不同类型的动画,除了包括了前面提到的 4 种基本的效果之外,还有一个混合了 scale 和 rotate 两种动画类型的效果,首先来看一看 alpha 动画的 xml 文件内容:

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <set xmlns:android="http://schemas.android.com/apk/res/android"
03     android:interpolator="@android:anim/accelerate_interpolator">
04     <alpha android:repeatCount="0" android:repeatMode="restart"
05         android:fromAlpha="1.0" android:toAlpha="0.5" android:duration="1000" />
06     <alpha android:repeatCount="0" android:repeatMode="restart"
07         android:fromAlpha="0.5" android:toAlpha="1.0" android:duration="1000"
08         android:startOffset="1000" />
09 </set>
```

其中,第 04～08 行定义了这个透明动画效果的执行方法,包括了:

- repeatCount——重复次数。
- repeatMode——重复方式。
- fromAlpha——动画起始的透明度,1.0 为完全不透明,0 为完全透明。

- toAlpha——动画终止的透明度。
- duration——动画过程完成的时间。
- startOffset——动画起始的时间。

可以看到这个动画效果由两个依次发生的动画效果组成，其中第 1 段动画是将某个 View 由不透明变为半透明的状态，第 2 段动画则是反过来变成不透明的状态，每一段动画的持续时间都是 1000ms。

另外，代码第 03 行的 interpolator 属性，本意是插值，简单地说就是动画过程中一些变量的变化方式，例如 accelerate_interpolator 就是一个带加速度变化的动画，在该段动画中的含义就是其变为透明或不透明的过程不是匀速进行的，而是越来越快地进行。有关 interpolator 更详细的信息将在 7.2.3 节中介绍。

rotate 动画的 xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<set>
    <rotate xmlns:android="http://schemas.android.com/apk/res/android"
        android:interpolator="@android:anim/decelerate_interpolator"
        android:duration="2000" android:pivotX="50%" android:pivotY="50%"
        android:fromDegrees="0" android:toDegrees="-720">
    </rotate>
</set>
```

定义旋转动画的方式与透明动画类似，这里需要额外提到的两个属性是：

- pivotX——旋转轴 X 坐标相对于 View 左上角的位置，50%即为 X 方向上中点的位置。
- pivotY——旋转轴 Y 坐标相对于 View 左上角的位置，50%即为 Y 方向上中点的位置。

该段动画的效果即是 View 绕自己的几何中心点在 2000ms 的时间内由 0 度旋转至 −720 度的位置。

scale 动画的 xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <scale android:interpolator="@android:anim/decelerate_interpolator"
        android:duration="2000" android:fromXScale="0.0" android:toXScale="1.0"
        android:fromYScale="0.0" android:toYScale="1.0" android:pivotX="0%"
        android:pivotY="100%" android:fillAfter="true">
    </scale>
    <scale android:interpolator="@android:anim/decelerate_interpolator"
        android:duration="2000" android:fromXScale="0.0" android:toXScale="1.0"
        android:fromYScale="0.0" android:toYScale="1.0" android:pivotX="50%"
        android:pivotY="50%" android:fillAfter="true">
    </scale>
</set>
```

其中：

- fillAfter——当该值为 true 时，动画将会停止在最后一帧，与之相对应的另一个属性

是 fillBefore，当 fillBefore 值为 true 时，动画将会恢复到第一帧停止。

translate 动画的 xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate android:interpolator="@android:anim/decelerate_interpolator"
        android:duration="1000" android:fromXDelta="0" android:toXDelta="300"
        android:fromYDelta="0" android:toYDelta="0">
    </translate>
    <translate android:interpolator="@android:anim/decelerate_interpolator"
        android:duration="1000" android:startOffset="1000" android:fromXDelta="0"
        android:toXDelta="-300" android:fromYDelta="0" android:toYDelta="0">
    </translate>
</set>
```

最后，混合了 scale 和 rotate 的动画效果 xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <scale android:interpolator="@android:anim/accelerate_interpolator"
        android:duration="1000" android:fromXScale="1.0" android:toXScale="0.6"
        android:fromYScale="1.0" android:toYScale="0.6" android:pivotX="50%"
        android:pivotY="50%" />
    <rotate android:interpolator="@android:anim/accelerate_interpolator"
        android:duration="1000" android:pivotX="50%" android:pivotY="50%"
        android:fromDegrees="0" android:toDegrees="-45" />
</set>
```

补间动画可以应用到任何 View 上，为了便于观察，选择了在图片按钮（ImageButton）上来运行动画效果，为此，在 Activity 的布局文件中依次添加了 5 个图片按钮，另外还添加了一个按钮用于同时触发所有的 5 个动画，代码略。

剩下的工作就是要将每一个图片按钮关联上一个动画效果，使得可以通过单击图片按钮来触发相应的动画，Activity 的代码如下：

```
public class TweenAnimationActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        final ImageButton alphaAnimationButton = (ImageButton)
            findViewById(R.id.alphaButton);
        final ImageButton rotateAnimationButton = (ImageButton)
            findViewById(R.id.rotateButton);
        final ImageButton scaleAnimationButton = (ImageButton)
            findViewById(R.id.scaleButton);
        final ImageButton translateAnimationButton = (ImageButton)
              findViewById(R.id.translateButton);
```

```
        final ImageButton compositeAnimationButton = (ImageButton)
                findViewById(R.id.compositeButton);
        final Button activateAll = (Button) findViewById(R.id.activateAll);
        final Animation alphaAnimation = AnimationUtils.loadAnimation(
                getApplicationContext(), R.anim.alpha_animation);
        final Animation rotateAnimation = AnimationUtils.loadAnimation(
                getApplicationContext(), R.anim.rotate_animation);
        final Animation scaleAnimation = AnimationUtils.loadAnimation(
                getApplicationContext(), R.anim.scale_animation);
        final Animation translateAnimation = AnimationUtils.loadAnimation(
                getApplicationContext(), R.anim.translate_animation);
        final Animation compositeAnimation = AnimationUtils.loadAnimation(
                getApplicationContext(), R.anim.composite_animation);
        //让动画停止在最后一帧,反之则会恢复到第一帧
        compositeAnimation.setFillAfter(true);
        compositeAnimation.setFillBefore(false);

        OnClickListener buttonClickListener = new OnClickListener() {

            @Override
            public void onClick(View v) {
                switch (v.getId()) {
                case R.id.alphaButton:
                    alphaAnimationButton.startAnimation(alphaAnimation);
                    break;
                case R.id.rotateButton:
                    rotateAnimationButton.startAnimation(rotateAnimation);
                    break;
                case R.id.scaleButton:
                    scaleAnimationButton.startAnimation(scaleAnimation);
                    break;
                case R.id.translateButton:
                    translateAnimationButton.startAnimation(translateAnimation);
                    break;
                case R.id.compositeButton:
                    compositeAnimationButton.startAnimation(compositeAnimation);
                    break;
                case  R.id.activateAll:
                    alphaAnimationButton.startAnimation(alphaAnimation);
                    rotateAnimationButton.startAnimation(rotateAnimation);
                    scaleAnimationButton.startAnimation(scaleAnimation);
                    translateAnimationButton.startAnimation(translateAnimation);
                    compositeAnimationButton.startAnimation(compositeAnimation);
                    break;
                default:
                    break;
                }
            }
        };

```

```
            alphaAnimationButton.setOnClickListener(buttonClickListener);
            rotateAnimationButton.setOnClickListener(buttonClickListener);
            scaleAnimationButton.setOnClickListener(buttonClickListener);
            translateAnimationButton.setOnClickListener(buttonClickListener);
            compositeAnimationButton.setOnClickListener(buttonClickListener);
            activateAll.setOnClickListener(buttonClickListener);
        }
    }
```

示例的运行效果如图 7-3 所示。

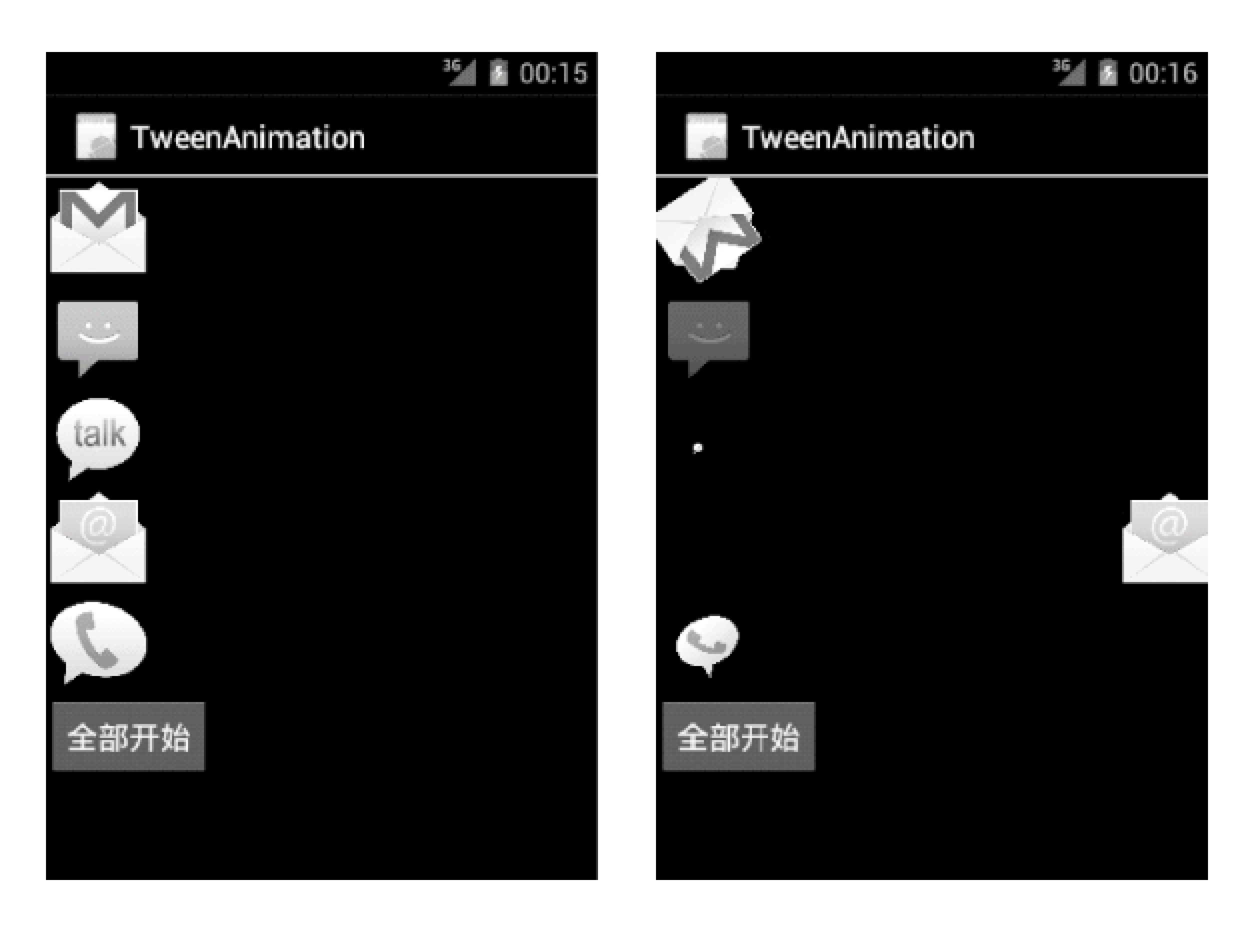


图 7-3 补间动画效果图

7.2.3 属性动画系统(Property Animation System)

Property Animation System(以下简称 PAS)可以将动画(Animation)应用到几乎所有的对象之上,这里的动画不仅仅包括视觉上的动画效果,任何对象的任何属性值按照一定规律进行变化的过程都可以被理解为"动画",通常所说的动画只不过是将这些属性值的变化(例如尺寸、旋转角度、位置等变化)表现到了视觉效果上而已,为了表述清晰,后面将这些在动画中进行变化的属性值统称为"动画值"。

要实现动画效果,除了指定特定的某种属性作为对象之外,还需要确定如何来"运行"这个动画,例如,对该对象的位置进行变换、动画所持续的时间长短、动画的初始状态和结果状态等。PAS 为这种动画的实现提供了一个比较完善的框架,通常你仅仅需要定义如下几种特征就能够确立出一个动画效果:

- 持续时间——为动画效果定义其持续的时间长短,默认值是 300ms。
- 时间插值(Time interpolation)——这个特征定义了如何根据动画当前已经持续的时间来计算出现在的属性值,例如,线性的时间插值使得属性值随时间线性变化(如匀速运动的对象),而非线性的时间插值则可以使得属性值随时间呈非线性变化(如在动画过程初期加速运动而在末期减速运动的对象)。
- 动画循环的次数及行为——通过这个特征可以决定是否重复地进行某个动画效果,或者你也可以使得某个动画效果反向地进行。

- 动画效果集——可以将多个独立的动画效果复合起来，让它们同时执行，或者线性执行，或者一个接一个的按照一定的延时来进行。
- 每帧刷新的间隔——定义更新一帧所等待的时长，这通常需要参考系统当前的运行速度来决定，否则过短的刷新间隔可能会导致系统过于繁忙。

1. PAS 相关 API 简介

下面通过表 7-1～表 7-3 来了解 PAS 提供的 API。

表 7-1　代表动画的类

类	简　介
ValueAnimator	这个类作为运行动画的引擎，并且它也负责计算出动画中属性变化的值。这个类包括了其所代表的动画效果的详细信息
ObjectAnimator	它是 ValueAnimator 的子类。相当于对 ValueAnimator 进行了封装，使用它可以更容易在对象的层次上来施加动画效果，因此通常情况下会使用这个类，如果要在这个类的某个属性上添加动画效果，那么至少为该属性提供一个 setter 方法，并且这个方法名需要与属性名严格对应，例如属性名为 propName，那么它的 setter 方法名应该为 setPropName()，否则将会出现找不到方法的错误，通常也建议为这个属性添加一个 getter 方法，getter 方法将在没有为动画设定初始状态的情况下被使用
AnimatorSet	动画效果集合，将多个独立的动画效果复合起来，让它们同时执行，或者线性执行，或者一个接一个地按照一定的延时来进行

表 7-2　用于计算属性值的类

类或接口	简　介
IntEvaluator	为整型属性计算值
FloatEvaluator	为浮点型属性计算值
ArgbEvaluator	为颜色属性计算值
TypeEvaluator	所有用于计算属性值的类都需要实现这个接口，利用这个接口可以实现用于计算自定义类型属性的值

表 7-3　代表时间插值方式的类

类或接口	简　介
AccelerateDecelerateInterpolator	在动画的初期和末期进行缓慢的加速和减速，在动画的中期速度最快
AccelerateInterpolator	动画在初期变化缓慢，然后逐渐加速变化
AnticipateInterpolator	动画在初期先向反方向变化，然后再向正方向加速变化
AnticipateOvershootInterpolator	类似于上一种方式，在动画末期会超过最终的值然后再逐渐恢复到最终值
BounceInterpolator	动画的末期被处理成弹跳的方式
CycleInterpolator	使动画循环执行
DecelerateInterpolator	动画在初期快速变化，然后逐渐低速变化
LinearInterpolator	以一定的变化率匀速变化
OvershootInterpolator	动画快速地到达并超过最终值，然后再恢复
TimeInterpolator	这是一个接口，用于开发人员实现自己的 TimeInterpolator

2．图解

本节第 1 部分介绍了 PAS 相关的诸多类和接口，然而仅仅从文字上进行描述显得非常空洞，因为动画毕竟是一个连续变化的过程，受限于书本的表现形式，本节通过图片来进行较形象的描述，读者可以借助于 API demos 提供的示例（运行 API Demos→Views→Animation→Interpolators）来进一步了解这些动画。

设想一个使对象的位置在某个方向上发生变化的动画效果，如果使用 LinearInterpolator 类来实现这个动画，那么对象位置发生的变化与时间的关系将会如图 7-4 所示。

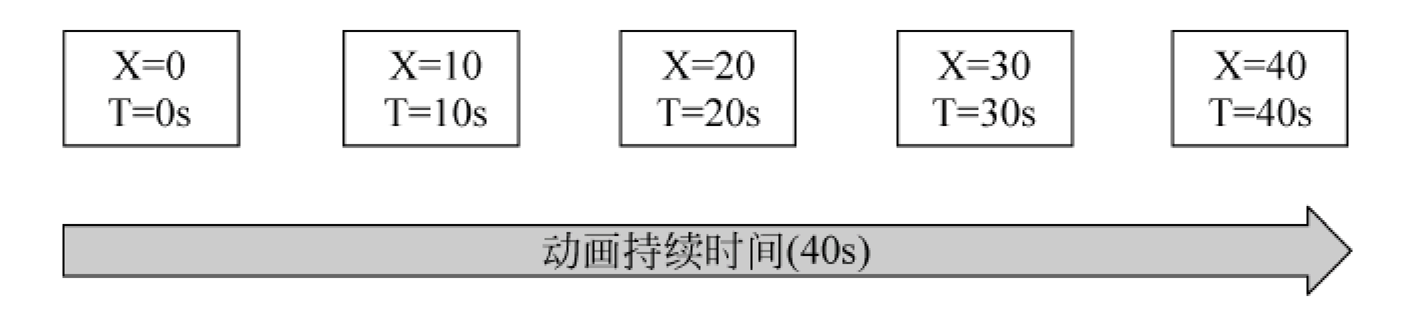


图 7-4　匀速变化的动画效果示意

如果使用 AccelerateDecelerateInterpolator 类来实现这个动画，那么对象位置发生的变化与时间的关系将会如图 7-5 所示。

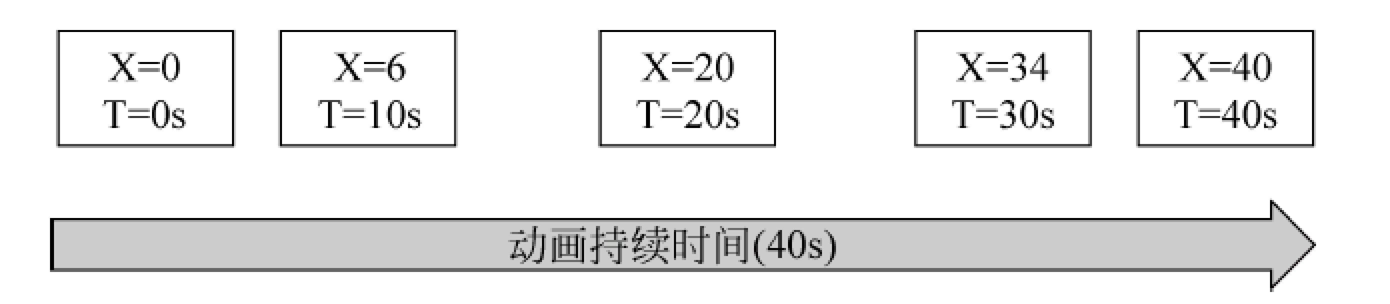


图 7-5　带加速度变化的动画效果示意

整个动画效果的实现通常会涉及若干个组件，即通过表 7-1～表 7-3 中的各个类配合完成，从根本上来说，就是依靠这些类来对动画值进行计算，然后将动画值的变化反映到视觉效果上，从而实现动画效果。这里以 ValueAnimator 为例，各组件之间的关系如图 7-6 所示（该图改自 Android 官方开发文档）。

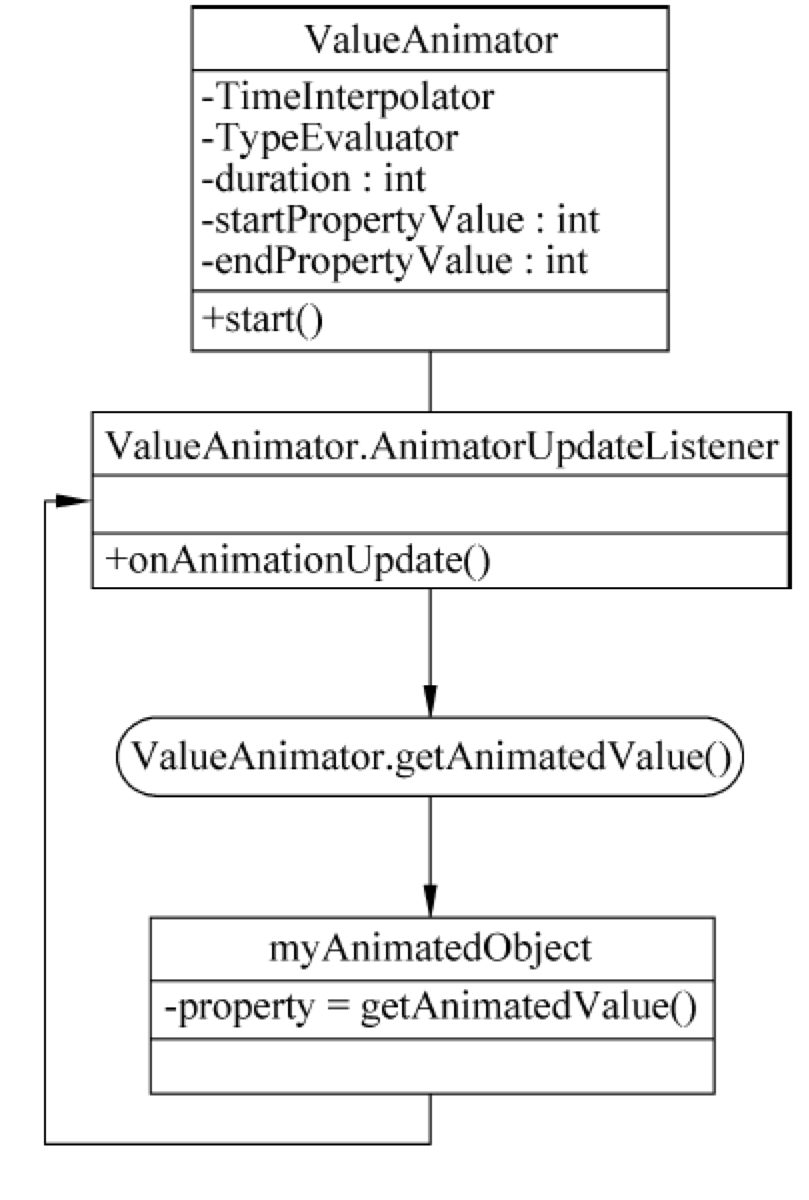


图 7-6　动画效果中的求值过程

如图 7-6 所示，最上方的是代表 ValueAnimator 类的框图，可以看到一个 ValueAnimator 主要包括了用于确定动画值的变化方式的 TimeInterpolator 对象、用于具体计算实时的动画值的 TypeEvaluator、动画持续的时间 duration、动画值的起始值以及最终值，并且还提供了一个 start()方法用于启动一次动画过程。

图 7-6 中第二个框图代表 ValueAnimator. Animator UpdateListener 接口，每个动画对象都应该注册一个实现了该接口的监听器，这个监听器所包含的 onAnimationUpdate()方法将在动画值更新后立即被

调用，在该方法中进行一些处理从而实现对动画状态的更新。而动画值的更新则由TimeInterpolator和TypeEvaluator来完成，每当这两个对象计算出一个新的动画值，就将调用onAnimationUpdate()方法，在onAnimationUpdate()方法中通常会调用invalidate()方法使界面重绘，从而使用ValueAnimator.getAnimatedValue()方法来获取当前最新的动画值，并根据这个最新的动画值来运行动画。

3. 动画效果示例

本部分将利用PAS来实现一些简单的动画，以便读者掌握如何使用PAS来设计动画效果。

为了便于展示动画效果，首先实现了一个ShapeHolder类，它的实例代表着用于运行动画效果的对象(例如圆形、矩形)，这个类包含了一些属性，并且提供了这些属性的getter/setter方法，从而使得动画效果能够成功地运用到这些属性上(见表7-1中对ObjectAnimator的描述)，ShapeHolder类的代码如下：

```
//用于保存对象的形状以及用于绘制该对象的属性,还提供了这些属性的getter/setter方法
public class ShapeHolder {
    private float x = 0, y = 0;
    private ShapeDrawable shape;

    public ShapeHolder(ShapeDrawable s) {
        shape = s;
    }

    public void setX(float value) {
        x = value;
    }
    public float getX() {
        return x;
    }

    public void setY(float value) {
        y = value;
    }
    public float getY() {
        return y;
    }

    public ShapeDrawable getShape() {
        return shape;
    }

    public float getWidth() {
        return shape.getShape().getWidth();
    }
    public void setWidth(float width) {
        Shape s = shape.getShape();
        s.resize(width, s.getHeight());
    }
```

```

    public float getHeight() {
        return shape.getShape().getHeight();
    }
    public void setHeight(float height) {
        Shape s = shape.getShape();
        s.resize(s.getWidth(), height);
    }
}
```

如上面的代码所示，属性 x，y 分别代表了对象在屏幕上的坐标，而 shape 则代表了对象的形状，通过对 x 和 y 属性施加动画，就能够使得这个对象在屏幕上按相应的动画效果进行运动。

接下来实现用于显示动画的 Activity，动画效果以 View 对象（这里的 View 对象名为 MyAnimationView）作为载体，因此只需要在实现了这个 MyAnimationView 之后将其加入到该 Activity 的 Layout 即可，向 Layout 中添加 View 可以通过如下代码来实现：

```
setContentView(R.layout.main);
LinearLayout container = (LinearLayout) findViewById(R.id.container);
final MyAnimationView animView = new MyAnimationView(this);
container.addView(animView);
```

如上面的代码所示，只需要获取到一个 ViewGroup 对象（这里是一个 LinearLayout），然后使用 addView()方法将新实现的 View 添加到这个 ViewGroup 中即可。

在实现了 ShapeHolder 类之后，需要在 Activity 中来使用这个类，具体的做法是实现一个方法用于够构造出所需的这个 ShapeHolder 的形状（shape），在本例中构造了一个具有渐变效果（gradient）的圆形对象，该方法仅需要两个参数代表圆心，从而用于计算该球形的初始坐标（x，y）：

```
    private ShapeHolder createBall(float x, float y) {
        OvalShape circle = new OvalShape();
        circle.resize(50f, 50f);
        ShapeDrawable drawable = new ShapeDrawable(circle);
        ShapeHolder shapeHolder = new ShapeHolder(drawable);
        shapeHolder.setX(x - 25f);
        shapeHolder.setY(y - 25f);
        int red = (int)(Math.random() * 255);
        int green = (int)(Math.random() * 255);
        int blue = (int)(Math.random() * 255);
        int color = 0xff000000 | red << 16 | green << 8 | blue;
        Paint paint = drawable.getPaint();
        int darkColor = 0xff000000 | red/4 << 16 | green/4 << 8 | blue/4;
        RadialGradient gradient = new RadialGradient(37.5f, 12.5f,
                50f, color, darkColor, Shader.TileMode.CLAMP);
        paint.setShader(gradient);
        return shapeHolder;
    }
```

如上面的代码所示，首先通过第 02 行和第 03 行代码构造了一个直径 50 像素的圆形，然后使用这个圆形构造出 ShapeHolder 对象（第 04 行和第 05 行），通过 setX()和 setY()方法来设置该对象在屏幕上的位置，第 08～16 行则使用随机产生的颜色值来生成了一个渐变的效果，并使用这个渐变效果来为圆形着色，从而使得对象具有球体的效果。

要在 MyAnimationView 中添加一个这样的球体，只需要为其添加一个 ShapeHolder 类型的对象，然后重写 onDraw()方法对这个球体进行绘制即可。MyAnimationView 的完整代码如下：

```
01 public class MyAnimationView extends View implements
02         ValueAnimator.AnimatorUpdateListener {
03     ShapeHolder ball = null;
04     ObjectAnimator animation = null;
05 
06     public MyAnimationView(Context context) {
07         super(context);
08         ball = createBall(25,50);
09     }
10 
11     @Override
12     public void onAnimationUpdate(ValueAnimator animation) {
13         invalidate();
14     }
15 
16     private void createAnimation() {
17         if (animation == null) {
18             ObjectAnimator myAnimator = ObjectAnimator.ofFloat(ball, "y",
19                     25f, getHeight() - ball.getHeight() - 25).setDuration(1000);
20             myAnimator.setInterpolator(new AnticipateOvershootInterpolator());
21             myAnimator.addUpdateListener(this);
22             animation = myAnimator;
23         }
24     }
25 
26     public void startAnimation() {
27         createAnimation();
28         animation.start();
29     }
30     //此处省略该方法内容
31     private ShapeHolder createBall(float x, float y) {
32     }
33 
34     @Override
35     protected void onDraw(Canvas canvas) {
36         canvas.save();
37         canvas.translate(ball.getX(), ball.getY());
38         ball.getShape().draw(canvas);
39         canvas.restore();
40     }
41 }
```

如上面的代码所示，MyAnimationView 继承自 View 并且实现了 AnimatorUpdateListener 接口，第 03 行和第 04 行分别定义了用于代表球体的 ShapeHolder 类对象 ball 和用于代表动画效果的 ObjectAnimation 对象 animation；第 06～09 行为 MyAnimationView 的构造方法，可以看到在其构造方法中使用了前面已介绍的 createBall()方法创建了一个球体对象并赋给了 ball；第 12～14 行则是实现了 AnimatorUpdateListener 接口方法，在接口方法中调用了 invalidate()方法来使得视图被重绘；第 35～40 行重写了 View. onDraw()代码使得 ball 能够被绘制到视图中。

代码中加粗的部分则是用于实现动画效果，其中第 16～24 行的 createAnimation()方法中实现了具体的代表动画效果的 animation 对象，此处使用了 AnticipateOvershootInterpolator，即动画在初期先向反方向变化，然后在向正方向加速变化，并且在动画末期会超过最终的值然后再逐渐恢复到最终值。读者可以对该方法内代码进行修改，从而实现各种不同的动画效果；startAnimation()方法则为外界提供启动这段动画的接口。

最后，为 Activity 添加一个按钮控件用于启动一次动画，然后在 Activity 的 onCreate 方法中添加如下代码即可：

```
Button starter = (Button) findViewById(R.id.startButton);
starter.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        animView.startAnimation();
    }
});
```

该示例动画的运行效果截图如图 7-7 和图 7-8 所示，由于只能截取几个状态并不能够完整的呈现动画效果，因此建议读者在模拟器或真机上实际地运行该示例来进行观察。

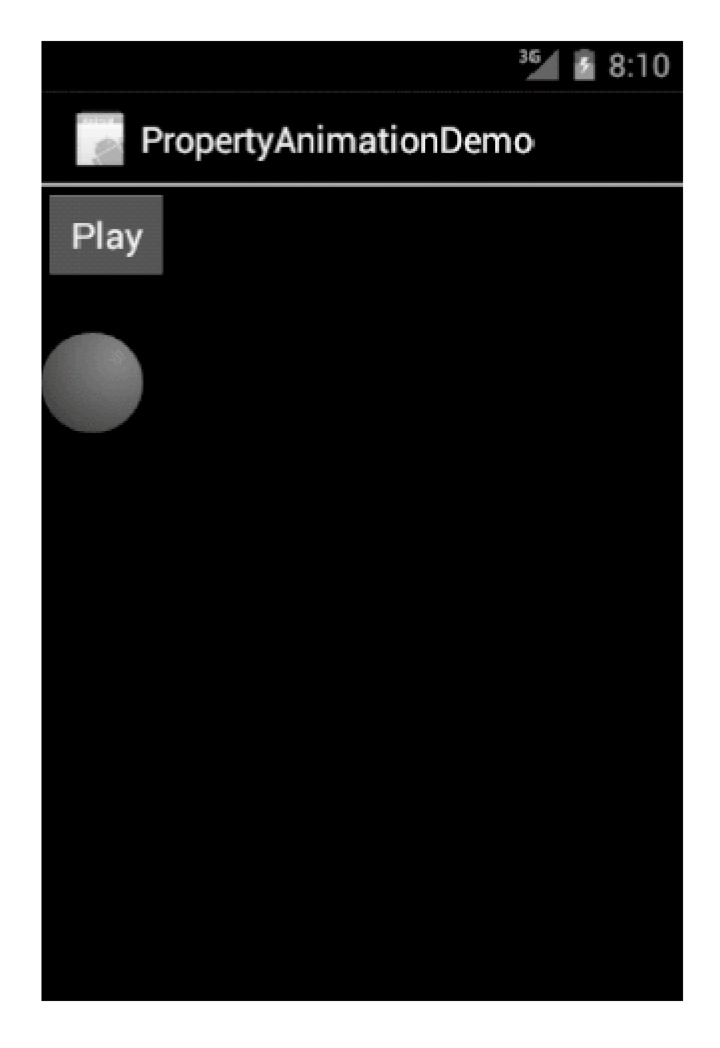


图 7-7　动画状态一

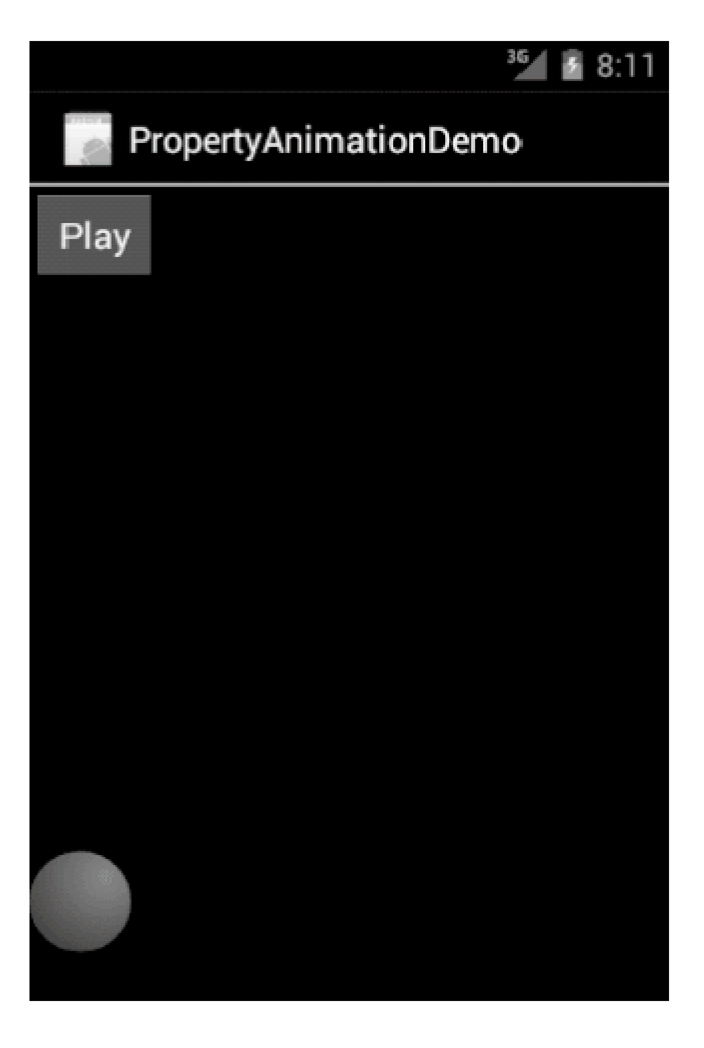


图 7-8　动画状态二

7.3 双缓冲技术

7.3.1 双缓冲技术简介

在图形图像处理中，双缓冲是一项应用十分普遍的技术，双缓冲技术的出现主要是为了防止屏幕在有大量数据需要绘制的情况下出现的闪烁现象。屏幕闪烁的原因是：当有大量的动画数据需要绘制并且需要频繁更新时，当前一帧尚未绘制完成时，后一帧又请求更新屏幕，这会使得屏幕被擦除并且使用背景色来填充绘图区域，由于背景色与新绘制图像之间的颜色反差，就会使得屏幕出现闪烁现象，而使用了双缓冲技术之后，可以将一帧图像先在内存中的缓冲区内绘制完成，然后再将这个缓冲区内容的图像通过一定的方法绘制到屏幕上，从而避免闪烁现象的发生。

双缓冲技术会在内存中创建一个与屏幕绘图区域一致的缓冲区，在生成图像时，首先会将图像绘制到内存中的这个缓冲区上，然后再一次性地将这个缓冲区内保存的图形复制到屏幕绘图区域，从而大大加快了绘图的速度，这个过程大致可以分为如下几步：

- 在内存中划分出一块与屏幕绘图区域一致的区域作为缓冲区。
- 将需要绘制的图像首先在缓冲区上进行绘制。
- 将缓冲区中绘制完成的图像数据复制到代表屏幕绘图区域的画布上。
- 释放内存中的缓冲区，或者按照上述步骤进行下一帧的绘图过程。

7.3.2 Android 中的双缓冲技术

在 Android 中，图像的显示主要通过 View 类来实现，View 类的子类 SurfaceView 的实现就采用了双缓冲技术，因此在进行游戏开发或者一些有大量图形处理的应用开发时，应该尽量使用 SurfaceView 而不是 View，例如，通常在绘制 3D 图形时就是用了 GLSurfaceView 类（SurfaceView 的子类）。

为了让读者更加直观地了解 SurfaceView 类中使用到的双缓冲技术，下面通过一个示例来进行详细说明，示例项目名称为 DoubleBufferingDemo。

该项目由一个 DoubleBufferingActivity 类和一个 MySurfaceView 类组成，其中 MySurfaceView 类是 SurfaceView 的子类并且实现了 SurfaceHolder.Callback 接口，本例主要考察的就是这个类的显示机制，在 MySurfaceView 类中包含了具体的绘制视图的代码，作为 SurfaceView 的子类通常需要实现如下几个方法：

```
public void surfaceCreated(SurfaceHolder holder) {}
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {}
public void surfaceDestroyed(SurfaceHolder holder) {}
```

这 3 个方法用于对 SurfaceView 的生命周期中的几个状态进行管理，另外：

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {}
```

该方法可以向 SurfaceView 所处的容器类提供自身的尺寸信息，具体的绘制视图的代码则通常置于 onDraw()方法中。

MySurfaceView 类的代码如下：

```
public class MySurfaceView extends SurfaceView implements Callback {
    private DrawingThread drawingThread;   //绘制 SurfaceView 的线程
    private int x = 0;                     //当前要绘制文字的横坐标
    private int theAmplitudeOfSin = 60;    //本例将文字绘制成按正弦曲线排列,这是振幅
    private final static int interval = 10; //两次绘制的横坐标间隔
    private Paint paint;

    public MySurfaceView(Context context) {
        super(context);
        getHolder().addCallback(this);
        drawingThread = new DrawingThread(this, getHolder());
        paint = new Paint();
        paint.setColor(Color.WHITE);
    }

    private class DrawingThread extends Thread{
    //该线程类用于实时地计算和更新下一帧需要绘制的内容,将在后面给出
    }

    public void step(){
        drawingThread.stepThread();
    }

    public void onDraw(Canvas canvas){
        //当横坐标超过了模拟器的显示宽度(320)后,清除画布,
        //由于 SurfaceView 使用了双缓冲,因此此时只能清除掉两个 canvas 中的一个
        if(x >= 320){
            x = 0;
            canvas.drawColor(Color.BLACK);
        }

        //在清除了一个 canvas 后紧接着的下一次绘制操作时(即 x = 10),这时再次
        //执行清除画布的操作,可以清除掉双缓冲使用到的另一个 canvas 上的内容
        if(x == interval){
            canvas.drawColor(Color.BLACK);
        }

        //按照正弦关系,根据当前横坐标 x 的值计算出纵坐标 y,从而确定将要绘制到的坐标点
        int y = (int)(theAmplitudeOfSin * Math.sin(x/160.0f * Math.PI));
        canvas.drawText(x + "", x, y + theAmplitudeOfSin + 10, paint);
        //System.out.println("AsinSurfaceView 在坐标 x:" + x + ", y:" + y + "进行了绘制");
    }

    public void surfaceCreated(SurfaceHolder holder) {
        drawingThread.start();
    }

    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height){
```

```
49      }
50
51      public void surfaceDestroyed(SurfaceHolder holder) {
52          drawingThread.cancelThread();
53      }
54
55      @Override
56      protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
57          super.onMeasure(widthMeasureSpec, heightMeasureSpec);
58          setMeasuredDimension(320, 160);
59      }
60  }
```

上面代码段中的第 24～42 行就是用于进行绘制的 onDraw()方法，可以看到该方法的作用就是根据当前 x 的取值，利用正弦函数求得当前对应的 y 值，然后把当前的 x 值绘制到坐标点(x,y)处(第 39 行和第 40 行)，另外第 27～30 行和第 34～36 行，则是判定在达到边界值时对画布进行清空，为什么会使用两个 if 语句对画布清空两次，具体的原因参见代码中的注释，读者可以尝试将第 34～36 行的代码注释掉再运行示例，看看会出现什么样的效果，在更加详细地对双缓冲进行解释之前，先看一看绘图线程类 DrawingThread 的具体实现：

```
private class DrawingThread extends Thread{
    private MySurfaceView mySurfaceView;  //待绘制的 SurfaceView
    private SurfaceHolder surfaceHolder;  //对应 SurfaceView 的控制对象
    private boolean drawingFlag = true;   //是否处于绘图阶段
    private boolean stepFlag = false;     //是否进行下一帧绘图

    public DrawingThread(MySurfaceView asinSurfaceView,
            SurfaceHolder surfaceHolder){
        this.mySurfaceView = asinSurfaceView;
        this.surfaceHolder = surfaceHolder;
    }
    //绘制下一帧
    public void stepThread(){
        stepFlag = true;
    }

    //销毁绘制线程
    public void cancelThread(){
        drawingFlag = false;
        stepFlag = true;
    }

    @Override
    public void run() {
        while(drawingFlag == true){
            Canvas canvas = null;
            try{
                //绘制当前一帧
                canvas = surfaceHolder.lockCanvas(null);
                if(canvas != null){
                    onDraw(canvas);
```

```
32                      }
33                  }catch(Exception e){
34                  }finally{
35                      if(canvas != null){
36                          surfaceHolder.unlockCanvasAndPost(canvas);
37                      }
38                  }
39                  //更新需要绘制文字的横坐标
40                  mySurfaceView.x += interval;
41
42                  //如果没有收到绘制下一帧的命令,则一直等待
43                  while(!stepFlag);
44                  stepFlag = false;
45              }
46              super.run();
47          }
48      }
```

在上面的代码中，第 04 行和第 05 行定义了两个布尔类型的变量 drawingFlag 和 stepFlag，用于控制该绘图线程的状态；第 13～15 行和第 18～21 行代码为外部类提供了两个控制接口，即 stepThread()（单步执行绘制操作）和 cancelThread()（终止线程），这两个控制接口内部实际上就是对 drawingFlag 和 stepFlag 进行赋值操作；第 24～47 行则是该线程的 run()方法，run()方法内部首先包含了一个大的 while 循环，该循环执行的条件是 drawingFlag == true，在该循环内部首先完成当前一帧的绘制（第 26～38 行），通过 surfaceHolder.lockCanvas()方法来获取 SurfaceView 持有的当前需要绘制的 Canvas 对象，在绘制完毕之后再使用 surfaceHolder. unlockCanvasAndPost(canvas)方法来提交绘制结果。此处之所以采用了带异常捕获的 try-catch-finally 结构，是为了避免当该示例应用被切换到后台时所产生的异常，因为当程序在后台运行时，不能够获取 canvas 对象；在完成了一帧的绘制之后，代码第 40 行对 x 值进行了改变操作，即增加了一个值为 interval 的增量；第 43 行和第 44 行则是用于控制线程的单步执行。

DoubleBufferingActivity 的代码如下：

```
public class DoubleBufferingActivity extends Activity {
    MySurfaceView mySurfaceView;
    Button stepOver;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mySurfaceView = new MySurfaceView(this);
        stepOver = new Button(this);
        stepOver.setText("下一帧");

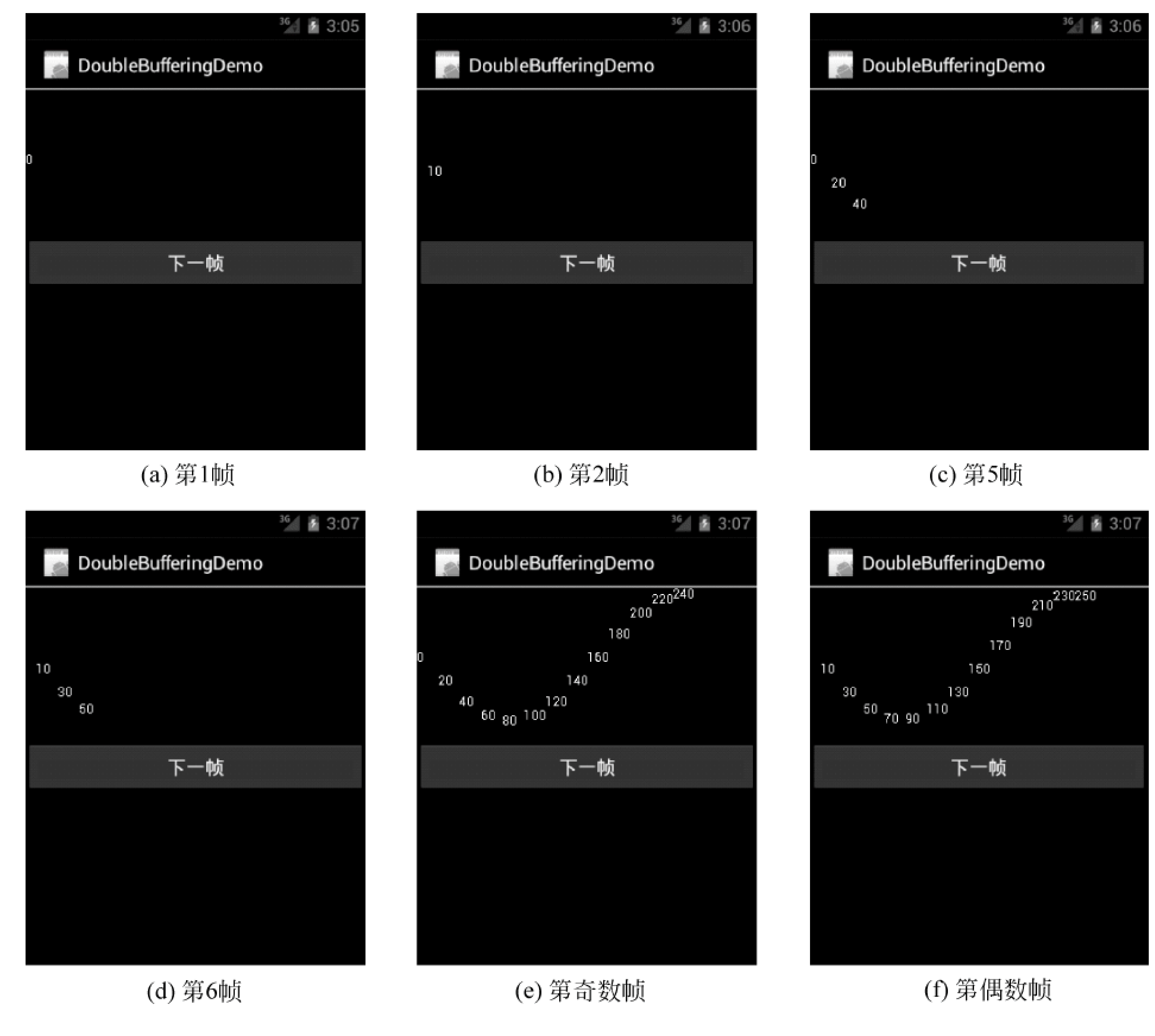
        LinearLayout l = new LinearLayout(this);
        l.setOrientation(LinearLayout.VERTICAL);
        l.addView(mySurfaceView);
        l.addView(stepOver);
        setContentView(l);

```

```
18          stepOver.setOnClickListener(new OnClickListener() {
19
20              @Override
21              public void onClick(View v) {
22                  mySurfaceView.step();
23              }
24          });
25      }
26  }
```

通常，如果要在 Activity 中使用到自己实现的 View，那么采取上面代码的方式（直接使用代码来将视图添加到 Activity）相对要比通过布局 xml 文件的方式要简便一些。代码第 08～16 行的作用就是以代码的形式来定义 Activity 布局界面，依次向界面中添加了一个之前实现的 MySurfaceView 对象和一个用于单步执行的按钮，第 18～24 行为按钮添加了单击事件监听器，通过调用 MySurfaceView 的 step()方法来实现单步执行绘制过程的功能。

此处将绘制线程控制为单步执行正是为了让读者能够看清 SurfaceView 的双缓冲技术的实现原理。运行示例，通过单击“下一帧”按钮，观察界面中每一帧的变化，这里随机截取的几个状态的截图如图 7-9 所示。



(a) 第1帧　(b) 第2帧　(c) 第5帧

(d) 第6帧　(e) 第奇数帧　(f) 第偶数帧

图 7-9　示例的 6 个状态

从上述状态可以清晰地了解到双缓冲的实现原理，其中奇数序号的截图与偶数序号的截图中所包含的数字都是不相同的，这些数字的绘制都是使用相同的方法(onDraw())来实现的，产生上列截图差异的原因就是因为双缓冲绘图时使用到了两个不同的画布(Canvas)，这两个画布是交替进行绘制和显示的：最开始屏幕上显示的是画布1的内容，而onDraw()方法将下一帧内容首先绘制到处于内存中的画布2上，当画布2上的内容绘制完成之后，交换画布1和画布2的位置，使得屏幕上显示出画布2上的内容，之后再下一帧的内容则绘制到画布1上，以此类推。这就是此处双缓冲技术的应用原理。

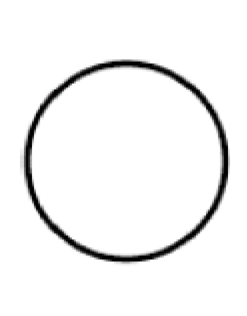
有关双缓冲技术的更多知识，已经超出了本书讨论的范围，请有兴趣的读者结合本节的内容，到互联网上或者其他的相关书籍中获取更多的信息。

7.4 使用 Path 类绘制 2D 图形

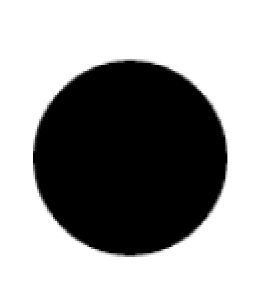
7.4.1 Path 类介绍

Android 提供的 Path 类可以用于绘制 2D 图形，简单地说，这个 Path 类的作用就是用于封装一系列由线段(方形、多边形等)、2次曲线(圆、椭圆等)和3次曲线复合而成的几何轨迹集合的类，借助于 Canvas 的 drawPath(path, paint)方法可以将这些轨迹集合绘制出来，这个方法所包含的另外一个参数 paint 可以被理解为用于绘制这些集合轨迹的“画笔”，这个画笔所具有的一些特性能够决定被绘制出的轨迹样式，例如当使用一个 paint 对象来绘制一个圆时，如果该 paint 的风格(style)为 filled(通过 setStyle(Paint. Style. FILL)方法设置)，则会绘制出一个实心圆；而如果其风格为 stroke(Paint. Style. STROKE)，则会绘制一个空心圆，并且这个空心圆的线宽可以使用 setStrokeWidth()方法来进行设置。3种不同风格的 paint 画出的圆形的效果如图 7-10 所示。这3种的风格依次是 STROKE、FILL、FILL_AND_STROKE。

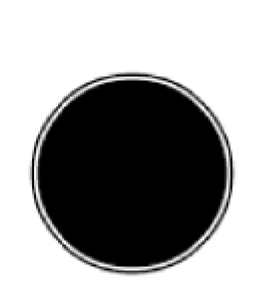
另外，Path 类也包含了两个比较常用的枚举类型，即 Path. Direction 和 Path. FillType，Path. Direction 指定了 Path 轮廓的绘制方向(用于闭合的图形，例如椭圆、矩形等)，而 Path. FillType 则指定了 Path 的填充方法。Path. Direction 比较好理解，仅仅包含 CCW(逆时针)和 CW(顺时针)两个值，分别对应于如图 7-11 所示的两种效果；而 Path. FillType 包含了4个不同的值，这里以两个相交的圆形为例，这4种 FillType 分别对应了如图 7-12 所示的4种效果，从左上角到右下角依次对应了 WINDING、EVEN_ODD、INVERSE_WINDING、INVERSE_EVEN_ODD 所代表的 FillType。



(a) STROKE



(b) FILL



(c) FILL_AND_STROKE

图 7-10 3种不同风格的 paint 的绘制效果



(a) 逆时针CCW



(b) 顺时针CW

图 7-11 Path. Direction 的两种情况

Path 提供了很多绘制基本图形的方法，例如：

- addArc()——用于绘制一段圆弧。
- addCircle()——用于绘制圆形。
- addOval()——用于绘制椭圆。
- addPath()——用于绘制一组轨迹。
- addRect()——用于绘制矩形。
- addRoundRect()——用于绘制圆角矩形。

可以使用这些现成的方法为基础通过组合绘制出各种图形。另外 Path 还提供了一些直接操作元线段的方法，例如 moveTo()、lineTo()等方法。

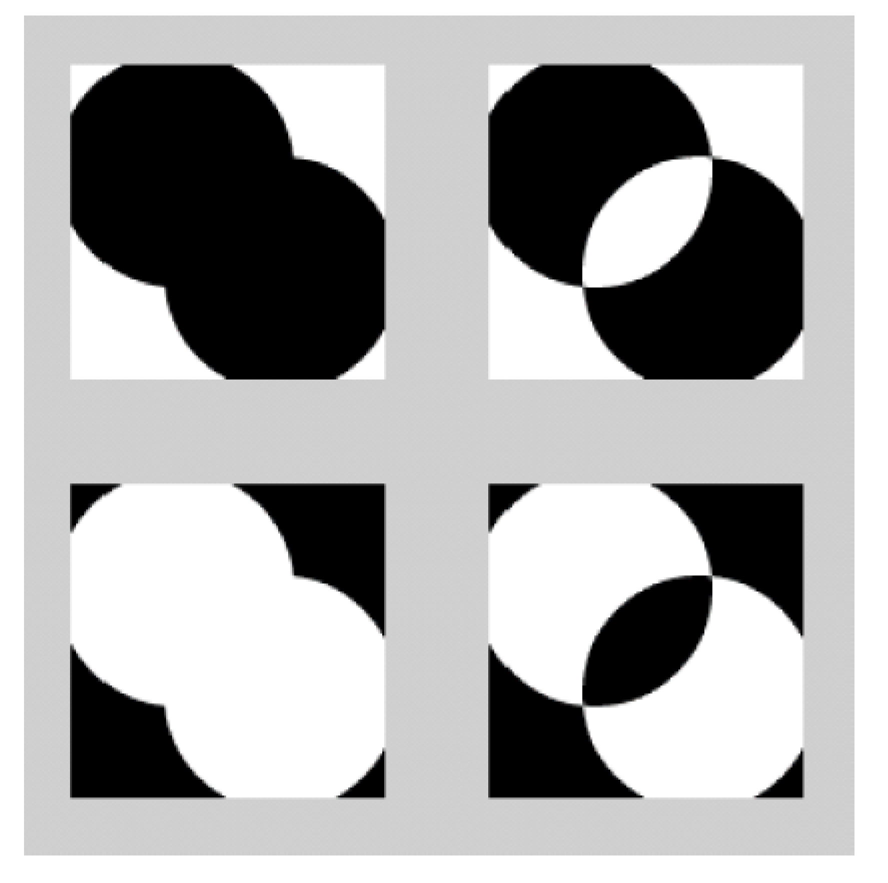


图 7-12　Path. FillType 的 4 种情况

下面通过示例(配套项目为 PathDraw)来说明如何使用 Path 类，配合 Canvas、Paint 等类绘制出一些最基本的图元，例如点、直线段和正圆等几何图形，最后实现通过触摸屏幕绘制一系列的点的功能。

7.4.2 触摸画点

为了实现能够在一块特定的区域内进行绘图，首先实现了一个用于绘制图像的视图类 PathView，然后在该类中实现用于绘制具体图像的方法，PathView 的代码如下：

```
public class PathView extends View {
    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private Path mPath;
    private boolean mCurDown;              //指示屏幕是否被按下(点击)
    private int mCurX;                     //被点击点的坐标
    private int mCurY;
    private final Rect mRect = new Rect();     //该矩形用于控制局部更新区域,用于触摸
                                               //画点功能

    public PathView(Context context, AttributeSet attrs) {
        super(context, attrs);
        setFocusable(true);
        setFocusableInTouchMode(true);
        mPaint.setStyle(Paint.Style.FILL_AND_STROKE);
        mPaint.setStrokeWidth(1);
        mPath = new Path();
    }
    private void showPath(Canvas canvas, int x, int y, Path.FillType ft,
                          Paint paint) {
        canvas.translate(x, y);
        canvas.clipRect(0, 0, 280, 300); //可以绘制图形的区域
        canvas.drawColor(Color.WHITE);   //为该区域着色
        mPath.setFillType(ft);           //设置 fill 方式
        canvas.drawPath(mPath, paint);   //绘制 Path
    }

    @Override protected void onDraw(Canvas canvas) {
```

```
27          Paint paint = mPaint;
28          canvas.drawColor(0xFFCCCCCC);     //为 Canvas 着色
29          paint.setAntiAlias(true);         //抗锯齿
30          paint.setColor(Color.BLACK);      //设置画笔颜色
31          paint.setPathEffect(null);        //不采用特效
32          showPath(canvas, 0, 0, Path.FillType.WINDING, paint);
                                              //绘制 Path,使用并集的 fill 方式
33      }
34  }
```

上面的代码已经实现了一个用于绘制图像的框架，相当于已经搭建起了一个画板。首先，第 02 行和第 03 行分别定义了画笔对象 mPaint 和用于存储图像的 mPath 对象；其次，第04～07 行则是与触摸画点功能相关的变量；再次，第 09～16 行是 PathView 的构造方法，构造方法中对 mPaint 对象进行了设置，并且实例化了 mPath 对象；最后，第 17～24 行的 showPath()方法则是在 canvas 上画出 mPath 所包含的一系列轨迹，showPath()方法由 onDraw()方法调用，而 onDraw()方法则将在视图需要更新时被系统自动调用。

如果读者去查阅 Path 类的 API，可以发现 Path 类并没有提供直接用于画点的方法，这是由于一个点实际上可以被看做是一个直径为 1 像素的圆形，因此直接使用 Path 类提供的 addCircle()方法即可，为此，在 PathView 类中实现一个名为 drawPoint 的方法，用于在某个坐标点处画一个点：

```
private void drawPoint(float x, float y) {
    if (mCurDown) {         //添加当前被触摸点并刷新视图显示
        mPath.addCircle(mCurX, mCurY, 1, Path.Direction.CCW);
        mRect.set((int)x - 3, (int)y - 3, (int)x + 3, (int)y + 3);
        invalidate(mRect); //只更新矩形区域
    }
}
```

实现了 drawPoint()方法后，每调用一次该方法，就可以在对应位置画出一个点。这里要实现的功能是通过触摸屏幕这个操作来在被触摸的位置画出一个点，为此，需要重写 View.onTouchEvent()方法，在该方法中添加实现触摸画点功能的代码：

```
    @Override
    public boolean onTouchEvent(MotionEvent event) {
    int action = event.getAction();          //获取事件类型
    if(action == MotionEvent.ACTION_DOWN || action == MotionEvent.ACTION_MOVE){
        mCurDown = true;
    }
    else{
        mCurDown = false;
    }
    int N = event.getHistorySize();          //获取历史被触摸点集合的元素个数
    for (int i = 0; i < N; i++) {            //依次添加历史被触摸点
        drawPoint(event.getHistoricalX(i), event.getHistoricalY(i));
    }
    drawPoint(event.getX(), event.getY());  //画出当前被触摸的点
```

```
15          return true;
16      }
```

其中，第 03～09 行判断是否为需要画点的触摸事件，第 10～13 行代码的功能则是画出所有被触摸过的点。因此，该示例也能够通过触摸画出一系列的点，但是目前的处理方式会使得这些点表现为断断续续的状态，读者可以在上述代码的基础上进行修改，使得程序拥有画出折线、平滑曲线的功能。

触摸画点的效果如图 7-13 所示。



图 7-13 触摸画点的效果

7.4.3 画线段

在实现触摸画点功能的过程中已经基本完成了一个简单的绘图框架，要画出一条线段就非常简单了，只需要将这条线段添加到前面的 mPath 变量中即可。在这里为了清晰说明如何使用 Path 类来绘制线段，示例实现的功能是：单击“画线段”按钮，在绘图界面上显示出一条固定的线段。了解了使用 Path 类画出线段的原理，读者可以尝试去实现类似于动态拖拽绘制线段或者点击两点绘制线段的功能。

在本例中绘制出一条固定线段使用的是 drawLine()方法，其代码如下：

```
01  //向 Path 中添加一条线段供绘制
02  public void drawLine(){
03  mPath.moveTo(20, 20);
04  mPath.lineTo(120, 120);
05  invalidate();
06  }
```

如上面的代码所示，要向 Path 中添加一条线段，仅仅需要类似于第 03 行和第 04 行这样的两行代码即可，其中第 03 行代码的作用是确定该条线段的一个端点，使用的是 Path 类的 moveTo()方法；第 04 行代码的作用就是指定该条线段的另一个端点并且向 Path 中添加一条以这两个坐标点为端点的线段。要绘制出这条线段，仅仅需要在“画线段”按钮的单击事件响应方法中调用 drawLine()方法即可：

```
Button drawLine = (Button)findViewById(R.id.button1);
drawLine.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        mPathView.drawLine();
    }
});
```

画出的线段如图 7-14 所示。

7.4.4　画其他几何图形



图 7-14　画线段效果

要绘制其他的几何图形，原理与前面画点和线段的原理基本相似，借助 Path 类提供的一些方法，可以绘制出各种各样的几何图形，例如，Path 类包含了如下一些方法：

- void addArc(RectF oval, float startAngle, float sweepAngle)——添加一段弧。
- void addCircle(float x, float y, float radius, Path.Direction dir)——添加一个圆。
- void addOval(RectF oval, Path.Direction dir)——添加一个椭圆。
- void addRect(float left, float top, float right, float bottom, Path.Direction dir)——添加一个矩形。
- void addRoundRect(RectF rect, float[] radii, Path.Direction dir)——添加一个圆角矩形。
- void arcTo(RectF oval, float startAngle, float sweepAngle)——类似于 lineTo，只是它用于添加一段弧。

利用 Path 所提供的这些方法，可以绘制出一些基本的 2D 图形。这里以绘制一个圆形为例，仍然使用前面所实现的绘图框架，类似于绘制线段，绘制圆的代码更加简单，仅仅需要调用 addCircle()方法，代码如下：

```
public void drawCircle(){//向 Path 中添加一个圆形供绘制
mPath.addCircle(180, 180, 90, Path.Direction.CCW);
invalidate();
}
```

画出的圆形如图 7-15 所示。

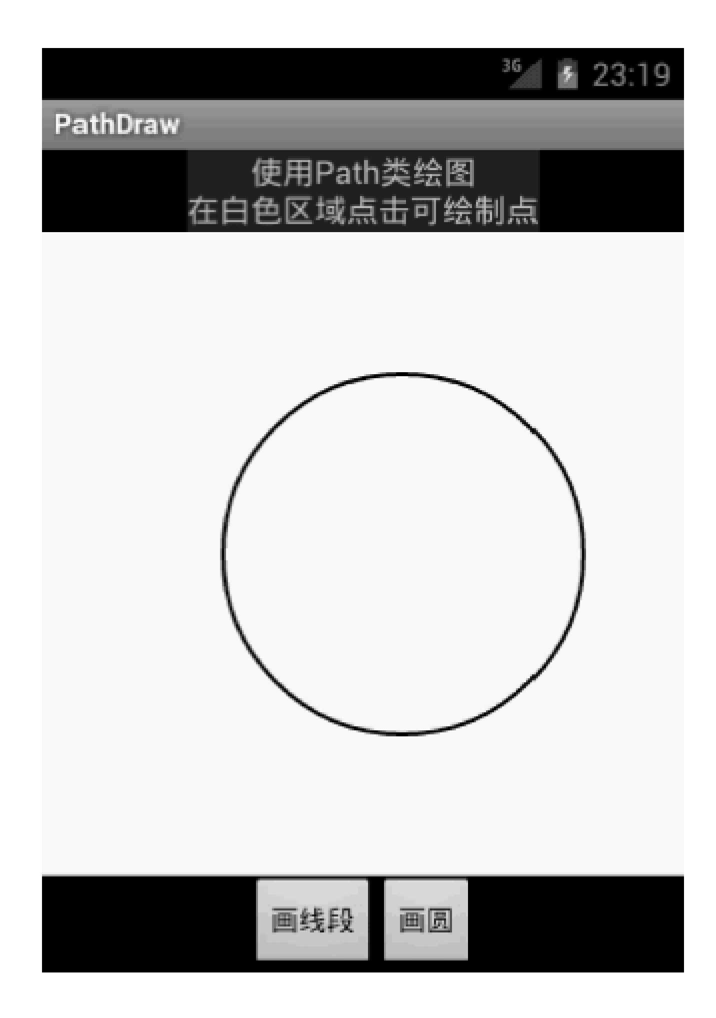


图 7-15　画圆效果

参考文献

1. Android 官方文档 Drawable Animation：http://developer. android. com/guide/topics/graphics/drawable-animation. html.
2. Android 官方文档 View Animation：http://developer. android. com/guide/topics/graphics/view-animation. html.
3. Android 官方文档 Property Animation：http://developer. android. com/guide/topics/graphics/prop-animation. html.
4. 百度百科“双缓冲技术绘图”词条：http://baike. baidu. com/view/1149326. htm.
5. Android 的 SurfaceView 双缓冲应用：http://www. oschina. net/code/snippet_54100_1422.
6. 自定义 VIEW 双缓冲与 SurfaceView 性能比较：http://www. eoeandroid. com/thread-68042-1-1. html.
7. Android ApiDemos 示例解析（70）：Graphics → PathFillTypes：http://www. imobilebbs. com/wordpress/?p=1589.

第8章 网络开发

当今世界是一个网络化信息化的世界，可以说人们每天都离不开网络。尤其是在移动通信设备如此普及的今天，无线网络通信的发展极为迅速，人们每天使用的短信、电话以及无线上网就是无线网络通信最基本的应用。可以毫不夸张地说，一台不支持无线网络连接的移动设备就像一台没有电视信号的电视机一样毫无价值。有了无线网络的支持，移动设备就可以不受空间的限制，这样才达到了真正的移动和自由，可以随时随地地接入互联网，浏览互联网上的最新资讯，与亲朋好友取得联系或者分析当前的股市进行投资。Android 作为一种优异的移动设备操作系统，提供了对网络通信的良好支持，如图 8-1 所示，作为开发人员，需要掌握它的网络通信开发技巧，以便开发出实用方便的网络应用。

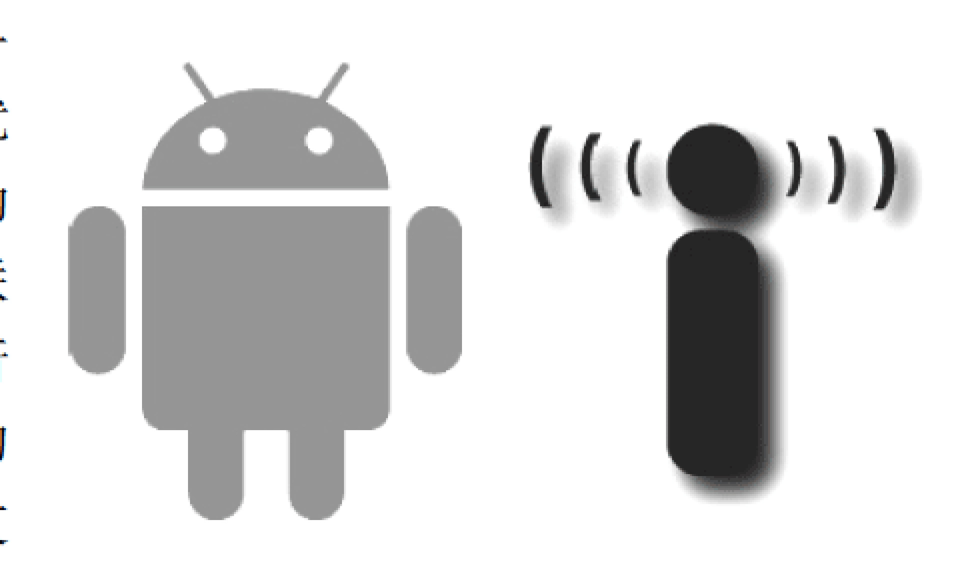


图 8-1　Android 与网络

在本章的开始先作一个说明：在 Android 中，如果要使用到网络相关的功能，都必须在应用的 AndroidManifest 文件中添加权限声明：

```
<uses-permission android:name="android.permission.INTERNET" />
```

8.1　网络通信支持

移动设备的网络功能的支持当然离不开硬件模块的支持，手机作为一种通信终端(MMS)，伴随着网络的升级而不断升级换代。1995 年 1G 问世，手机只能进行基本的语音通信，1996—1997 年 2G(GSM，CDMA)及其后的 GPRS、EDGE 等技术的快速发展，手机开始逐渐增加了数据服务功能。2009 年开始，3G 在全世界开始大规模布置以及苹果创造性开发新型苹果手机。手机慢慢变成了互联网的终端，从而开启了一个新的时代——移动互联网时代。因此现代手机通常都支持这些常用网络设备，包括 WIFI、NFC、蓝牙等模块。

Android 是由互联网巨头 Google 带头开发的，因此对网络功能的支持是必不可少的。Android 的应用层采用的是 Java 语言。所以 Java 支持的网络编程方式 Android 都能够很好地支持，同时 Google 还引入了 Apache 的 HTTP 扩展包。另外，针对 WIFI、NFC，分别提

供了单独的 API。在 Android 对各种型号的硬件的支持方面，由于 Android 已经做好了为上层应用提供服务的中间件，只要移动设备生产厂商按照 Android 系统的要求提供所需的硬件支持，Android 就能够使用这些硬件通信设备。Android 目前能够支持的网络通信模式有：GSM、Bluetooth、EDGE、3G、Wi-Fi 和近场通信(Near Field Communication)。

Android 网络开发包括如下的一些方式：

- 针对 TCP/IP 的 Socket、ServerSocket。
- 针对 UDP 的 DatagramSocket、DatagramPackage。
- 针对直接 URL 的 HttpURLConnection。
- Google 集成了 Apache HTTP 客户端，可使用 HTTP 进行网络编程。
- 使用 Web Service。
- 使用 WebView 视图组件，基于 WebView 进行开发。WebView 是 Android 提供的一个基于 chrome-lite 的 Web 浏览器，通过 WebView 就可以浏览网页。

本节首先对 Android 支持的一些通信技术进行介绍，然后在随后的章节中再对 Android 的网络开发方式进行介绍。

8.1.1 GSM

全球移动通信系统(Global System for Mobile Communications，GSM)如图 8-2 所示是当前应用最为广泛的移动电话标准。全球超过 200 个国家和地区超过 10 亿人正在使用 GSM 电话。GSM 标准的广泛使用使得在移动电话运营商之间签署“漫游协定”后用户的国际漫游变得很平常。GSM 较之它以前的标准最大的不同是它的信令和语音信道都是数字的，因此区别于第一代通信技术的模拟、仅限语音的蜂窝电话标准，GSM 被看作是第二代(2G)移动电话系统。GSM 标准当前由 3GPP(3rd Generation Partnership Project，第三代合作伙伴计划)组织负责制定和维护。



图 8-2 GSM 标志

从用户观点出发，GSM 的主要优势在于提供更高的数字语音质量和替代呼叫的低成本的新选择(比如短信)。从网络运营商的角度看来，其优势是能够部署来自不同厂商的设备，因为 GSM 作为开放标准提供了更容易的互操作性。而且，该标准允许网络运营商提供漫游服务，用户就可以在全球使用他们的移动电话了。

1. GSM 的历史

20 世纪 80 年代初，第一代移动电话技术开始应用，当时存在众多互不兼容的标准。仅在欧洲，就有北欧的 NMT、英国的 TACS、前西德等国使用的 C-450、法国的 Radiocom 2000 和意大利的 RTMI 等。用户的手机无法在其他标准的网络上使用，造成了很大的不便。由于这个原因，西欧国家开始考虑制定统一的下一代移动电话标准，以便能够提供更多样的功能和使用户漫游更加容易。最开始标准起草和制定的准备工作由欧洲邮电行政大会(CEPT)负责管理。具体工作由 1982 年起成立的一系列“移动专家组”负责。GSM 的名字即是移动专家组(法语：Groupe Spécial Mobile)的缩写。后来这一缩写的含义被改变为“全

球移动通信系统”,以方便GSM向全世界的推广。

1987年5月GSM成员国达成一致,确定了GSM最重要的几项关键技术。1989年,欧洲电信标准协会(ETSI)从CEPT接手标准的制定工作。1990年第一版GSM标准完成。1992年1月,芬兰的Oy Radiolinja Ab成为第一个商业运营的GSM网络。亚洲最早的GSM运营网络是香港电讯CSL(全称香港电讯Communication Services Limited,)香港电讯通讯服务有限公司。GSM的推出推动了移动通信的普及,用户持续快速增长。1995年,全球用户达到1千万,1998年达到1亿,2005年已经超过15亿。

1998年,目标为制订接替GSM的第三代移动电话(3G)规范的3GPP启动。3GPP也接受了维护和继续开发GSM规范的工作。ETSI是3GPP的成员之一。

在发展的过程中,GSM系统的功能不断得到丰富,从而能够提供更多样的服务。由GSM系统首先引入的短信息服务(SMS)提供了一种新颖、便捷、廉价的通信方式。1994年,GSM实现了基于电路交换的数据业务和传真服务。1999年,WAP协议使得用户可以通过手机访问互联网。2000年后开始商用的通用分组无线服务(GPRS)使得GSM系统能够以效率更高的分组方式提供数据通信。2003年,EDGE技术开始商用,提供了接近3G的数据通信能力。

目前,3GPP组织还在发展GSM标准,以便利用已经大量部署的GSM基础设施,平滑地向3G技术演进。

2. 市场状况

到2005年,全球有超过10亿人使用GSM电话,使GSM成为主导的移动电话系统,占到全球市场份额的70%。当前WCDMA并没有展现出全部的功能,而GSM的主要竞争对于CDMA2000(主要在北美、日本、中国和韩国使用)由于作为3G过渡的标准而获得了有限的增长。因为WCDMA网络建设已经起步(至少在高密度的市场),GSM正在缓慢消亡,但这将持续相当长的一段时间。

在1998—2000年之间导致GSM用户增长的主要原因是移动运营商推出预付费电话服务。它允许那些不能或者不想跟运营商签署合同的人们使用移动电话服务。这种服务在欧洲的移动运营商之间竞争也比较激烈,即使没有长期的合同,人们也可以从运营商那里以很低廉的价格买到一款手机。

3. GPRS

通用分组无线服务技术(General Packet Radio Service,GPRS)是GSM移动电话用户可用的一种移动数据业务。它经常被描述成“2.5G”,也就是说这项技术位于第二代(2G)和第三代(3G)移动通信技术之间。它通过利用GSM网络中未使用的TDMA信道,提供中速的数据传递。最初有人想通过扩展GPRS来覆盖其他标准,只是这些网络都正在转而使用GSM标准,这样GSM就成了GPRS唯一能够使用的网络。GPRS在Release 97之后被集成到GSM标准,在最开始它是由欧洲电信标准协会(European Telecommunications Standards Institute)负责标准化的,但是当前已经移交由3GPP负责。

1) GPRS原理

GPRS区别于旧的电路交换(或CSD)连接,连接在Release 97之前(GSM电话功能还

没怎么开发)就已经包含进 GSM 标准中。在旧有系统中一个数据连接要创建并保持一个电路连接,在整个连接过程中这条电路会被独占直到连接被拆除。GPRS 基于分组交换,也就是说,多个用户可以共享一个相同的传输信道,每个用户只有在传输数据的时候才会占用信道。这就意味着所有的可用带宽可以立即分配给当前发送数据的用户,这样更多的间隙发送或者接受数据的用户可以共享带宽。Web 浏览、收发电子邮件和即时消息都属于共享带宽的间歇传输数据的服务。

2) GPRS 的速率

通常 GPRS 数据的计费方式不是以秒,而是以千字节记。在电路交换方式下,即使网络上没有数据传输,其他用户也不能使用空闲的信道。基于 GPRS 的报文数据交换使用未使用的蜂窝网络带宽传输数据。作为专门为电话系统设计的语音信道(或者数据信道)一旦被报文数据交换使用,将降低可用带宽,其结果是如果在一个忙碌的电话域内,报文传输速度极慢。理论上报文数据交换速度是大约 170Kbps,而实际速度是 30～70Kbps。在 GPRS 的射频部分的改进,取名为 EDGE 技术,将支持从 20～200Kbps 的更高速度传输。最大数据速率取决于同时分配到的 TDMA 帧的时隙。因此,数据速率越高,纠错可靠性就越低。一般来说,连接速度随着距离的增加迅速下降。

4. EDGE

GSM 增强数据率演进(Enhanced Data rates for GSM Evolution,EDGE),是一种数字移动电话技术,作为一个 2G 和 2.5G(GPRS)的延伸,有时被称为 2.75G。这项技术工作在 TDMA 和 GSM 网络中。EDGE(通常又称为 EGPRS)是 GPRS 的扩展,可以工作在任何已经部署 GPRS 的网络上。

EDGE 在高速率的编码方案上使用 8 相位移相键控(8PSK)(每符号表示 3 比特信息)的调制技术。相对于 GSM 使用的高斯最小移位键控(GMSK)(每符号表示 1 比特信息),8PSK 每一个符号可以表示 3 比特的信息。这使得理论上 EDGE 能提供 3 倍于 GSM 的数据吞吐量。与 GPRS 一样,EDGE 使用速率匹配算法调整调制编码方案 (MCS),因此能保证无线信道、数据流量和数据传输的稳定。它引入了 GPRS 里没有的新技术:增加冗余度(Incremental redundancy)代替中继干扰报文发送更多的冗余信息来保持与接收机的联络,从而增加正确解码的概率。

在报文模式,它的最高数据数率是 384Kbps,因此符合 ITU 对 3G 网络的要求。作为 IMT-2000 家族的 3G 标准的一部分被国际电信联盟(ITU)接受。它还加强了电路交换模式,使用称为 HSCSD(High-Speed Circuit-Switched Data,高速电路交换数据)的技术提高数据交换速率。EDGE 大约在 2003 年最初由北美引入 GSM 网络。

8.1.2 3G

一般称为第三代移动通信技术(3rd-Generation,3G),也就是 IMT-2000(International Mobile Telecommunications-2000),是指支持高速数据传输的蜂窝移动通信技术。3G 服务能够同时传送声音(通话)及数据信息(电子邮件、即时通信等)。3G 的代表特征是提供高速数据业务,速率一般在几百 Kbps 以上。

3G 规范是由国际电信联盟(ITU)所制定的 IMT-2000 规范的最终发展结果。原先制

定的 3G 远景，是能够以此规范达到全球通信系统的标准化。目前 3G 存在 4 种标准：CDMA2000、WCDMA、TD-SCDMA 和 WiMAX。

3G 标准规定移动终端以车速移动时，数据传输速率为 144Kbps，室外静止或步行时速率为 384Kbps，在室内为 2Mbps。2007 年，ITU 制定了 IMT-A 标准，要求低速环境时峰值速率为 1Gbps，高速环境则是 100Mbps。

8.1.3 Wi-Fi

Wi-Fi 是由一个名为"无线以太网兼容联盟"(Wireless Ethernet Compatibility Alliance, WECA)的组织所发布的业界术语，其标志如图 8-3 所示。它是一种短程无线传输技术，能够在数百英尺范围内支持互联网接入的无线电信号。随着技术的发展，以及 IEEE 802.11a、IEEE 802.11g 等标准的出现，现在 IEEE 802.11 这个标准已被统称为 Wi-Fi。

Wi-Fi 这个术语是指无线保真(Wireless Fidelity)，类似历史悠久的音频设备分类：长期高保真(1930 年开始采用)或 Hi-Fi 的(1950 年开始采用)。即使 Wi-Fi 联盟本身也经常在新闻稿和文件中使用"无线保真"这个词，Wi-Fi 还是出现在 ITAA 的一篇论文中。

虽然 Wi-Fi 技术创建在 IEEE 802.11 标准上，但 IEEE 开发和出版这些标准，却不负责认证符合它的设备。于是非营利性的 Wi-Fi 联盟成立于 1999 年，用以填补这段空白，它的主要任务是创建和运行标准，维护互操作性与兼容性，并推动无线局域网技术。



图 8-3 Wi-Fi 的标志

截至 2009 年，Wi-Fi 联盟拥有超过 300 多个会员，这些会员来自世界各地公司和厂家，它们的产品在通过认证之后，有权在产品上标明 Wi-Fi 标志。认证过程具体来说就是认证产品是否符合 IEEE 802.11 无线标准的规定，包括了 WPA 和 WPA2 安全标准，以及 EAP 的认证标准。

目前 Wi-Fi 联盟所公布的认证种类有：

- WPA/WPA2——WPA/WPA2 是基于 IEEE 802.11a、802.11b、802.11g 的单模、双模或双频的产品所创建的测试程序。内容包含通信协定的验证、无线网络安全性机制的验证，以及网络传输表现与兼容性测试。
- WMM(Wi-Fi MultiMedia)——当影音多媒体通过无线网络的传递时，要如何验证其带宽保证的机制是否正常运作在不同的无线网络设备及不同的安全性设置上是 WMM 测试的目的。
- WMM Power Save——在影音多媒体通过无线网络的传递时，如何通过管理无线网络设备的待命时间来延长电池寿命，并且不影响其功能性，可以通过 WMM Power Save 的测试来验证。
- WPS(Wi-Fi Protected Setup)——这是一个 2007 年年初才发布的认证，目的是让消费者可以通过更简单的方式来设置无线网络设备，并且保证有一定的安全性。目前 WPS 允许通过 Pin Input Config(PIN)、Push Button Config(PBC)、USB Flash Drive Config(UFD)以及 Near Field Communication Contactless Token Config(NFC)的方式来设置无线网络设备。

- ASD(Application Specific Device)——这是针对除了无线网络访问点(Access Point)及站台(Station)之外其他有特殊应用的无线网络设备,例如 DVD 播放器、投影机、打印机等。
- CWG(Converged Wireless Group)——主要是针对 Wi-Fi mobile converged devices 的 RF 部分测量的测试程序。
- Wi-Fi Direct。
- 还有一种加密方式为 802.1x 用户认证机制,但目前兼容性不高。

1. Wi-Fi 的历史

IEEE 802.11 第一个版本发表于 1997 年,其中定义了介质访问接入控制层(MAC 层)和物理层。物理层定义了工作在 2.4GHz 的 ISM 频段上的两种无线调频方式和一种红外传输的方式,总数据传输速率设计为 2Mbps。两个设备之间的通信可以自由直接(ad hoc)的方式进行,也可以在基站(Base Station,BS)或者访问点(Access Point,AP)的协调下进行。

1999 年加上了两个补充版本:IEEE 802.11a 定义了一个在 5GHz ISM 频段上的数据传输速率可达 54Mbps 的物理层,IEEE 802.11b 定义了一个在 2.4GHz 的 ISM 频段上但数据传输速率高达 11Mbps 的物理层。

2.4GHz 的 ISM 频段为世界上绝大多数国家通用,因此 IEEE 802.11b 得到了最为广泛的应用。苹果公司把自己开发的 IEEE 802.11 标准起名叫 AirPort。1999 年工业界成立了 Wi-Fi 联盟,致力解决符合 IEEE 802.11 标准的产品的生产和设备兼容性问题。Wi-Fi 为制定 IEEE 802.11 无线网络的组织,并非代表无线网络。

2. Wi-Fi 的用途

1) 为移动设备提供小范围的局域无线网络连接

- 对于具备 Wi-Fi 功能的设备:如个人计算机、游戏机、智能手机或数字音频播放器可以从范围内的无线网络连接到网络。其覆盖范围的一个或多个(互联)接入点(也称之为热点)可以组成一个面积小到几间房间,或大到几平方英里的区域。
- 组织和企业:比如机场、酒店、餐馆等经常提供给来访者免费的热点,以吸引或协助客户。商家会依爱好者或希望提供服务,甚至以促进企业在某些领域有时会提供免费的 Wi-Fi 接入。目前在中国,许多大型的酒店和商场的内部,一般都会覆盖有免费开锁的 Wi-Fi 热点供来访者登录互联网。Wi-Fi 的(Muni-Fi)的项目已经开始,截止至 2008 年已有超过 300 个城市参与。2010 年,捷克共和国已有 1150 家 Wi-Fi 网服务供应商。
- 无线路由器:结合了数字用户线调制解调器或电缆调制解调器和 Wi-Fi 接入点,通常设置在家庭房屋、酒店客房或其他场所,可以提供互联网接入和互联网络的所有设备连接(无线或有线)。但因为家用无线路由器的功率较小,所以其信号覆盖范围、信号强度也很小。随着 MiFi 和 WiBro(便携式 Wi-Fi 路由器)的出现,可以很容易地创建自己的 Wi-Fi 热点,通过电信网络连接到网络。现在,许多移动电话(智能手机)也可以充当一个小型的无线路由器,供周围的设备接入互联网。新版本的

Android 就可以将自身模拟成为一个热点。借助这种方法可以使用 Ad hoc(没有有线基础设施支持的移动网络)模式为客户端到客户端连接 Wi-Fi 设备,而无须路由器,这种无线 Ad hoc 网络模式受到了大多数人的欢迎,掌上游戏机,如任天堂的 DS、数码相机和其他消费电子设备。同样,Wi-Fi 联盟正在推动一个新的发现和安全的方法规范,使得 Wi-Fi 可以直接用于文件传输和媒体共享。Wi-Fi 使得无线网络可以覆盖浴室、厨房和花园棚屋等任何区域,使得网络无所不在。

2) 城市 Wi-Fi 覆盖

在 21 世纪初期,世界各地许多城市宣布,计划全市范围的 Wi-Fi 网络。这被证明比最初发起人设想得更为困难,结果,大多数的这些项目都被取消或无限期搁置。但是有几个是成功的,例如在 2005 年,美国加州 Sunnyvale,成为第一个在美国的城市,提供全市免费无线网络连接。2010 年 5 月,在伦敦,英国伦敦市市长鲍里斯·约翰逊承诺 Wi-Fi 的普及;到 2012 年,无论是伦敦金融城或伊斯灵顿已经有广泛的室外 Wi-Fi 覆盖。全球已建和建造中的 Wi-Fi 城市已经超过 500 个,其中覆盖率最高的城市为中国台北市,全市已有 4000 个无线访问点(Access Point,AP),未来将增至 10 000 个,覆盖率达到 90%,全球主要的城市许多已有 Wi-Fi 技术,如伦敦、纽约、台北、香港、新加坡、汉堡、巴黎、华盛顿、上海等。

3) 校园 Wi-Fi 覆盖

卡内基梅隆大学于 1994 年在其匹兹堡校区创建了世界上第一个无线网络,比起源于 1999 年的 Wi-Fi 品牌还要早。现在大多数校园已具无线上网。在中国的许多大学图书馆内,也专门设有免费 Wi-Fi 热点,提供给学生使用。2000 年,费城德雷克塞尔大学创造了历史,成为美国第一个提供完全校园无线网络覆盖的大学。

3. Wi-Fi 的优势与局限

1) 优势

Wi-Fi 可以使得创建无线局域网(WLAN)的过程变得非常简单,它可让客户端设备无线连接到网络,采用这种方式通常可以降低网络部署和扩充的成本。在许多不能架设电缆的地区,如户外区域和历史建筑,就可以运用无线局域网来改善网络的覆盖。现在大多数笔记本计算机制造商都已经在产品中默认包含了连接无线网络的设备。随着支持 Wi-Fi 的设备的价位持续下跌,使得 Wi-Fi 技术越来越广泛地被使用,Wi-Fi 已经成为了企业普遍的基础设施。

在 Wi-Fi 标准的约束下,不同品牌的接入点和客户端所使用的网络接口之间能够实现互操作性,被 Wi-Fi 联盟认证并授权"Wi-Fi 认证"的产品都是向后兼容的,正是凭借 Wi-Fi 所制定的这一套全球统一标准,任何遵循 Wi-Fi 标准的设备可以在世界上任何地方无差异的工作,这是相对于移动电话技术的最明显的优势。目前 Wi-Fi 已使用在 22 万个以上的公共热点和几千万户家庭、公司及世界各地的大学校园中。当前 Wi-Fi 的 WPA2 密钥安全认证协议在 2010 年被业界广泛地认为是安全的,它可以为用户提供强大而安全的密码。

2) 局限

目前 Wi-Fi 还存在几个局限:

- 全球各地所执行的频谱分配和操作限制并不完全一致,例如在美国所使用的标准在 2.4GHz 频带分配了 11 个通道,而在欧洲大部分地区还分配有另外的 2 个通道,总

共 13 个通道，日本则还要多出一个通道。

- Wi-Fi 网络的覆盖范围目前还非常有限，一个典型的使用 IEEE 802.11b 或 IEEE 802.11g 标准的无线路由器的覆盖范围通常是 32m(室内)到 95m(室外)，而使用 IEEE 802.11n 标准的无线路由器所能覆盖的距离也只能达到前者的 2 倍左右。
- Wi-Fi 技术在短距离的传输上，与蓝牙等技术相比，其耗电量相对要高得多，这也使得 Wi-Fi 移动设备的电池寿命较短。

8.1.4 蓝牙

蓝牙(Bluetooth，如图 8-4 所示)是一种无线个人局域网(Wireless PAN)，它是一种开放式无线通信标准，提供设备短距离互联解决方案。蓝牙最初由爱立信创制，后来由蓝牙技术联盟制定技术标准。据说为了强调此技术及应用尚在萌芽阶段的意义，故将 Bluetooth 中文译名为较文雅的“蓝芽”，并在台湾进行商业的注册。在 2006 年，蓝牙技术联盟组织已将全球中文译名统一改采直译为“蓝牙”。目前蓝牙最新公布的版本是 4.1。



图 8-4 蓝牙的标志

蓝牙的标准是 IEEE 802.15.1，蓝牙协议工作在无须许可的 ISM(Industrial Scientific Medical)频段的 2.45GHz，最高速度可达 723.1Kbps。为了避免干扰可能使用 2.45GHz 的其他协议，蓝牙协议将该频段划分成 79 频道，(带宽为 1MHz)每秒的频道转换可达 1600 次。

拿蓝牙与 Wi-Fi 相比其实是不适当的，因为 Wi-Fi 是一种更加快速的协议，覆盖范围更广。虽然两者使用相同的频率范围，但是 Wi-Fi 需要更加昂贵的硬件。蓝牙设计被用来在不同的设备之间创建无线连接，而 Wi-Fi 是一种无线局域网协议。两者的目的是不同的。

1. 蓝牙的历史

蓝牙的名称来自 10 世纪丹麦国的蓝牙王哈拉尔德(Harald Gormsson)。出身海盗家庭的哈拉尔德统一了北欧四分五裂的国家，成为维京王国的国王，由于他喜欢吃蓝莓，牙齿常常染成蓝色，故得此号。用来暗示蓝牙是统一通信协议的通用标准。因为名称怪异的缘故，1998 年，爱立信公司希望无线通信技术能统一标准而取名“蓝牙”。

蓝牙技术最初由爱立信创制。技术始于爱立信公司的 1994 方案，它是研究在移动电话和其他配件间进行低功耗、低成本无线通信连接的方法。发明者希望为设备间的通信创造一组统一规则(标准化协议)，以解决用户间互不兼容的移动电子设备。1997 年前爱立信公司就此概念接触了移动设备制造商，讨论其项目合作发展，结果获得了支持。

1999 年 5 月 20 日，索尼爱立信、国际商业机器、英特尔、诺基亚及东芝公司等业界龙头创立“特别兴趣小组”(Special Interest Group，SIG)，即蓝牙技术联盟的前身，目标是开发一个成本低、效益高，可以在短距离范围内随意无线连接的蓝牙技术标准。

1998 年，蓝牙推出 0.7 规格，支持 Baseband 与 LMP(Link Manager Protocol)通信协定两部分。1999 年推出先后 0.8 版、0.9 版、1.0 Draft 版、1.0a 版、1.0B 版、1.0 Draft 版，完成 SDP(Service Discovery Protocol)协定、TCS(Telephony Control Specification)协定。1999 年 7 月 26 日正式公布 1.0 版，确定使用 2.4GHz 频谱，最高资料传输速度为 1Mbps，

同时开始了大规模宣传。与当时流行的红外线技术相比，蓝牙有着更高的传输速度，而且不需要像红外线那样进行接口对接口的连接，所有蓝牙设备基本上只要在有效通信范围内使用，就可以进行随时连接。

当 1.0 规格推出以后，蓝牙并未立即得到广泛应用，除了当时对应蓝牙功能的电子设备种类少，另一个原因是蓝牙装置也十分昂贵。2001 年的 1.1 版正式列入 IEEE 标准，Bluetooth 1.1 即为 IEEE 802.15.1。同年，SIG 成员公司超过 2000 家。过了几年之后，采用蓝牙技术的电子装置如雨后春笋般增加，售价也大幅回落。为了扩宽蓝牙的应用层面和传输速度，SIG 先后推出了 1.2 版、2.0 版，以及其他附加新功能，例如 EDR（Enhanced Data Rate，配合 2.0 的技术标准，将最大传输速度提高到 3Mbps）、A2DP（Advanced Audio Distribution Profile，一个控音轨分配技术，主要应用于立体声耳机）、ACRCP（A/V Remote Control Profile）等。Bluetooth 2.0 将传输率提升至 2Mbps、3Mbps，远大于 1.x 版的 1Mbps（实际约 723.2Kbps）。

较完整的版本发展如表 8-1 所示。目前广泛使用的是 3.0 版本，4.0 版本正在开发中。

表 8-1 蓝牙版本发展列表

版　本	规范发布日期	增 强 功 能
0.7	1998 年 10 月 19 日	Baseband，LMP
0.8	1999 年 1 月 21 日	HCI、L2CAP，RFCOMM
0.9	1999 年 4 月 30 日	OBEX 与 IrDA 的互通性
1.0 Draft	1999 年 7 月 5 日	SDP，TCS
1.0 A	1999 年 7 月 26 日	—
1.0 B	2000 年 10 月 1 日	WAP 应用上更具互通性
1.1	2001 年 2 月 22 日	IEEE 802.15.1
1.2	2003 年 11 月 5 日	列入 IEEE 802.15.1a
2.0 ＋ EDR	2004 年 11 月 9 日	EDR 传输率提升至 2～3Mbps
2.1 ＋ EDR	2007 年 7 月 26 日	简易安全配对，暂停与继续加密、Sniff 省电
3.0 ＋ HS	2009 年 4 月 21 日	交替射频技术，取消了 UMB 的应用
4.0 ＋ HS	2010 年 6 月 30 日	传统蓝牙技术、高速蓝牙和新的蓝牙低功耗技术

2. 蓝牙协议栈

蓝牙技术建立在一套协议栈之上（如图 8-5 所示），协议栈主要包括了如下几层：

- 核心协议层（基带、主控制层接口 HCI、链路管理协议 LMP、逻辑链路控制和适配协议 L2CAP、服务发现协议 SDP）。
- 线缆替换协议层（无线射频通信 RFCOMM）。
- 电话控制协议层（二进制电话控制标准 TCS-BIN、电话控制协议标准的 AT 命令集合）。
- 选用协议层（PPP，TCP，IP，UDP，OBEX，IrMC，WAP，WAE）。

1）核心协议层

基带层定义了蓝牙设备相互通信过程中必需的编解码、跳频频率的生成和选择等技术。

HCI Host Controller Interface 主控制器接口 在应用层（可选）为 LMP 和 Baseband 层

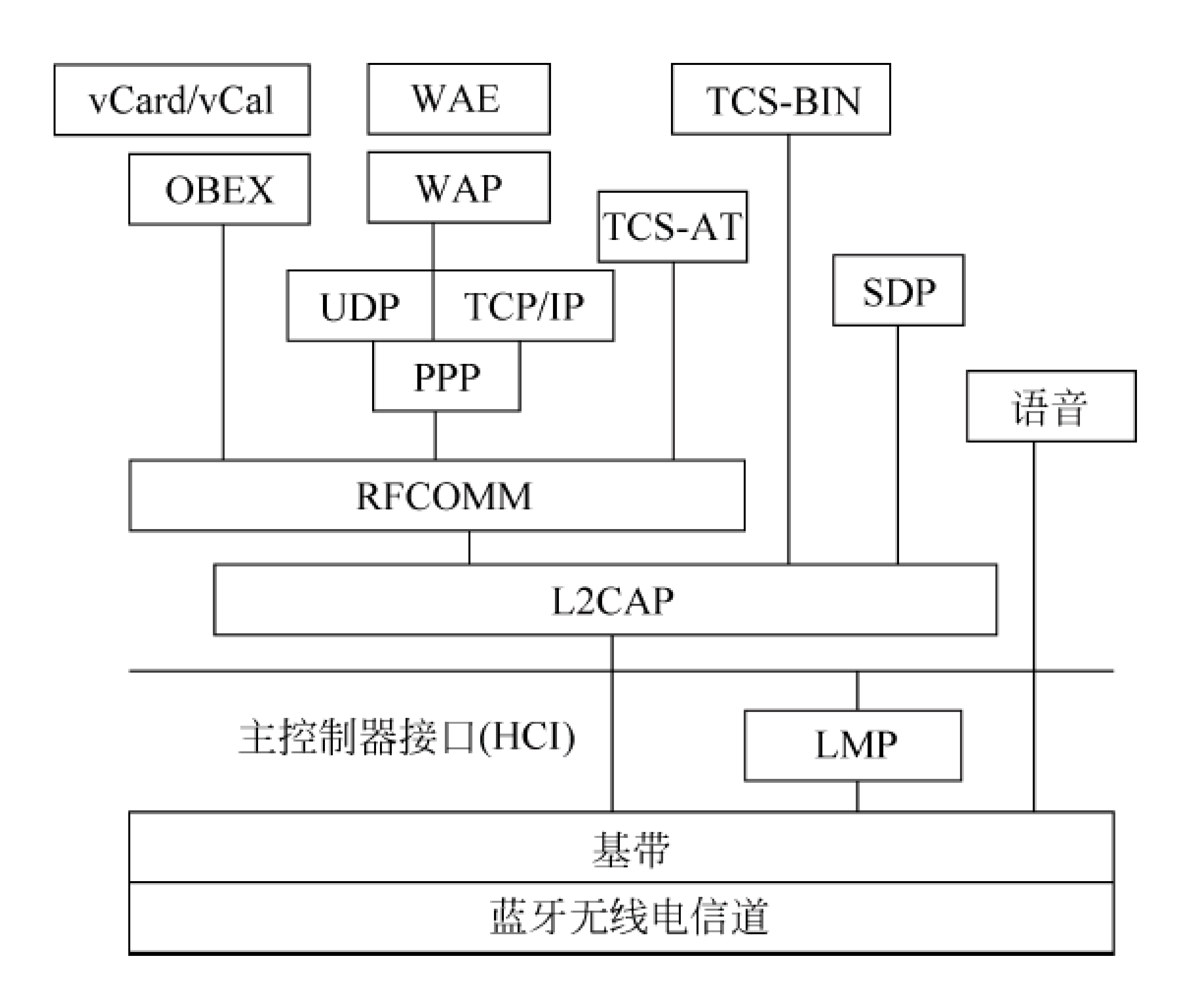


图 8-5 蓝牙协议栈

提供命令接口。

LMP Link Manager Protocol 链路管理协议 For LM peer to peer communication 用于链接设置和控制。LMP 协议数据单元(PDU)信号是通过接收方的链路管理器(Link Manager)进行解释和过滤的，并不传送到高层。LMP 的作用主要是完成基带连接的建立和管理。

L2CAP。逻辑链路控制和适配协议，负责适配基带中的上层协议。它同链路管理器并行工作，向上层协议提供定向连接的和无连接的数据业务。这个上层具有 L2CAP 的分割和重组功能，使更高层次的协议和应用能够以 64KB 的长度发送和接收数据包。它还能够处理协议的多路复用，以提供多种连接和多个连接类型(通过一个空中接口)，同时提供服务质量支持和成组通信。

业务搜索协议(SDP)包括一个客户/服务器架构，负责侦测或通报其他蓝牙设备。

2）线缆替换协议层

RFCOMM。一个基于欧洲电信标准协会 ETSI07.10 规范的串行线性仿真协议。此协议提供 RS232 控制和状态信号，如基带上的损坏、CTS 以及数据信号等，为上层业务(如传统的串行线缆应用)提供了传送能力。

3）电话控制协议层

TCS Binary 蓝牙电话控制协议标准，使用的是面向位的协议，也称为 TCS-BIN 系统。TCS-BIN 用于无绳电话规范。

TCS-AT 电话控制协议标准的一套 AT 命令集合，可在多种使用模式下控制移动电话和调制解调器。在英国电信的规定中，AT 命令基于 ITU-T 推荐的 v.250 和 ETS 300 916 (GSM 07.07)标准。除此之外，还规定了传真服务所用的命令。TCS-AT 也使用拨号网络和耳机规范。

4）选用协议层

- PPP：Point-to-Point Protocol，点对点协议。
- TCP/IP：传输控制协议/网间网协议。

- IrMC Ir Mobile Communications：红外移动通信。
- OBEX：OBject Exchange Protocol，对象交换协议。
- WAP：Wireless Application Protocol，无线应用协议。
- UDP：User Datagram Protocol，用户数据报协议。
- FTP：File Transfer Protocol，文件传输协议。

3. 蓝牙规范

蓝牙规范(profile)——为了保证蓝牙设备的互通性而制定的一系列蓝牙应用规范。蓝牙应用是根据蓝牙 SIG(特殊兴趣小组)所称的“应用规范”进行分类的，profiles 就是形成使用模式的规范。蓝牙应用规范的目的是为应用的互操作性提供构架。蓝牙应用规范根据某个协议所能支持的特殊使用模式或功能来对这个协议的消息和特性进行详细说明。使用模式包括传送、局域网访问、目标推进和同步。一些蓝牙应用规范仍在定义之中，这里列出了由 Bluetooth SIG(special interest group)制定的一部分 profile：

(1) 蓝牙立体声音频传输规范——蓝牙立体声音频传输规范(Advance Audio Distribution Profile，A2DP)，可播放立体声。

(2) 基本图像规范——基本图像规范(Basic Imaging Profile，BIP)在设备之间传送图像。可再细分成下列的组成：

- Image Push。
- Image Pull。
- Advanced Image Printing。
- Automatic Archive。
- Remote Camera。
- Remote Display。

(3) 基本打印规范——基本打印规范(Basic Printing Profile，BPP)可将文件、电子邮件传至打印机打印。

(4) 无线电话规范——无线电话规范(Cordless Telephony Profile，CTP)，蓝牙无线电话之间沟通的规范。

(5) 网内通信规范——网内通信规范(Intercom Profile，IP)，是另类的 TCS(Telephone Control protocol Specification)基底规范，是两个 Bluetooth 通信设备间沟通的规范。

(6) 调用网络规范——Baseband、LMP、L2CAP、SDP、RFCOMM 协议所需要的传输需求。

(7) 传真规范——传真规范(Fax Profile，FP)能传输传真的数据。

(8) 人机界面规范——人机界面规范(Human Interface Device Profile，HIDP)可支持鼠标、键盘功能。

(9) 蓝牙耳机规范——头戴式通话器规范(Headset Profile，HP)将声音传送到蓝牙耳机设备。

(10) 串行端口规范——串行端口规范(Serial Port Profile，SPP)用来取代有线的 RS-232 Cable。

(11) SIM 卡访问规范——SIM 卡访问规范(SIM Access Profile，SIM-AP)，可访问手

机内的 SIM 卡。

(12) 同步规范——同步规范(Synchronization Profile,SP),创建在 serial port profile、generic access profile 与 generic access profile 之上。

(13) 文件传输规范——文件传输规范(File Transfer Profile,FTP),Bluetooth 利用 OBEX 通信协议传送文件。

(14) 通用访问规范——通用访问规范(Generic Access Profile,GAP)用来创建连接。

(15) 通用对象交换规范——通用对象交换规范(Generic Object Exchange Profile,GOEP)使用 OBEX 进行对象交换。

(16) 对象交换规范——对象交换规范(Object Push Profile,OPP),Bluetooth 利用 OBEX 通信协议在两个设备间交换数据。

(17) 个人局域网规范——个人局域网规范(Personal Area Networking Profile,PANP)可支持蓝牙网络第三层协议。

(18) 电话簿访问规范——电话簿访问规范(Phone Book Access Profile,PBAP)可在设备之间互换电话簿。

(19) 图像分享规范——图像分享规范(Video Distribution Profile,VDP)可用来分享图像功能。使用 H.263 编码法。

(20) 服务发现应用规范——服务发现应用规范(Service Discovery Application Profile,SDAP)描述了一个应用程序如何使用 SDP(服务发现协议)来发现一个远程设备上运行的服务,SDAP 要求一个应用程序能够发现其所连接到的蓝牙设备上运行的所有服务。

读者可以在互联网上查询到更多的蓝牙规范及其说明。

4. 蓝牙的优点

蓝牙具有如下的一些优点:

- 蓝牙工作在全球开放的 2.4GHz ISM(即工业、科学、医学)频段。
- 使用跳频频谱扩展技术,把频带分成若干个跳频信道(hop channel),在一次连接中,无线电收发器按一定的码序列不断地从一个信道“跳”到另一个信道。
- 无须驱动程序,而是采用独特的配置文件。
- 一台蓝牙设备可同时与多台蓝牙设备建立连接,摆脱纠缠不清的电缆连接。
- 数据传输速率可达 1Mbps。
- 低功耗、低成本、通信安全性好、稳定。
- 在有效范围内可越过障碍物进行连接,没有特别的通信视角和方向要求。
- 支持语音传输。
- 组网简单方便,易于使用,即时连接。

5. 蓝牙的局限

较早版本的蓝牙技术存在着如下一些局限性:

- 传输速率有限,不适合传送过大的文件。
- 传输距离较短。
- 两个设备“握手”(handshaking)的过程中,蓝牙硬件的地址(BD_ADDR)会被传递出

去,在协定的层面上不能做到匿名,造成泄露数据的隐患。
- Bluetooth 在 2.4GHz 的电波干扰问题一直为大家所诟病,特别和无线局域网间的互相干扰问题。有干扰发生时,就以重新传送分组的方法来解决干扰。

6. 蓝牙的应用

利用蓝牙技术可以开发出很多相关的应用,例如:
- 移动电话和免提设备之间的无线通信,这也是最初流行的应用。
- 特定距离内计算机间的无线网络。
- 电脑与外设的无线连接,如鼠标、耳麦、打印机等。
- 蓝牙设备之间的文件传输。
- 传统有线设备的无线化。
- 数个以太网之间的无线桥架。
- 7 代家用游戏机的手柄,PS3、PSP go、Nitendo Wii。
- 依靠蓝牙支持使 PC 或 PDA 能通过手机的调制解调器实现拨号上网。
- 实时定位系统(RTLS),应用“节点”或“标签”嵌入被跟踪物品中读卡器从标签接收并处理无线信号以确定物品位置。

8.1.5 NFC

NFC(Near Field Communication,近场通信)也称近距离无线通信,是一种短距离的高频无线通信技术,允许电子设备之间进行近距离的(10cm 左右)非接触式点对点传输,其认证标志如图 8-6 所示。NFC 由非接触式射频识别(RFID)及互联互通技术整合演变而来,实现了在单一芯片上结合感应式读卡器、感应式卡片和点对点功能,能在短距离内与兼容设备进行设备和数据交换。NFC 在最初仅仅是对 RFID 技术和网络技术的简单合并,但是现在已经演变成为一种短距离无线通信技术,发展态势相当迅速。



图 8-6 NFC 认证标志

与 RFID 不同的是,NFC 具有双向识别和连接的特点,工作于 13.56MHz 频率范围,NFC 采取了独特的信号衰减技术,相对于 RFID 来说 NFC 具有距离近、带宽高、能耗低等特点。

1. NFC 的发展

NFC 的诞生可以追溯到 RFID 技术的出现。RFID 即射频识别技术,它允许一个阅读器通过发送电波的方式来获取标签上的信息,从而利用这个信息来对标签进行识别和追踪。在此列出一些 NFC 发展过程中的一些标志性事件:
- 1983 年,第一份以 RFID 为缩写的专利被授予了 Charles Walton。
- 2002 年,NXP Semiconductors 和 Sony 公司联合发明了 NFC 技术。
- 2004 年,Nokia、Philips 和 Sony 3 家公司建立了 NFC Forum,其官方页面地址为 http://www.nfc-forum.org。
- 2006 年,NFC 标签的初版规格被定义。

- 2006 年，Nokia 6131 成为了世界上第一部支持 NFC 的手机。
- 2009 年 1 月，NFC Forum 发布了 NFC 的端对端(Peer-to-Peer)标准，专用于传输手机联系人、URL、初始化蓝牙连接等功能。
- 2010 年，第一部支持 NFC 的 Android 手机 Samsung Nexus S 发布。
- 2011 年，Google I/O 大会上的"How to NFC"上展示了使用 NFC 初始化游戏和分享联系人、URL、应用程序、视频等数据。
- 2011 年，NFC 得到 Symbian Anna 的支持。
- 2011 年，RIM 作为第一家手机制造企业，其制造的设备被 MasterCard Worldwide 认证，从而真正使得手机支付变得便捷。

2. NFC 的工作模式

1) 卡模式(Card emulation)

这个模式其实就是相当于一张采用 RFID 技术的 IC 卡。可以替代现在大量的 IC 卡(包括信用卡)场合商场刷卡、公交卡、门禁管制、车票、门票等。此种方式下，有一个极大的优点，那就是卡片通过非接触读卡器的 RF 域来供电，不需要自带电源，即便是寄主设备(如手机)没电也可以工作。

2) 点对点模式(P2P mode)

这个模式和红外线差不多，可用于数据交换，只是传输距离比较短，传输创建速度快很多，传输速度也快些，功耗低(蓝牙也类似)。将两个具备 NFC 功能的设备连接，能实现数据点对点传输，如下载音乐、交换图片或者同步设备地址簿。因此通过 NFC，多个设备，如数字相机、PDA、计算机、手机之间，都可以交换资料或者服务。

3) 读卡器模式(Reader/writer mode)

作为非接触读卡器使用，比如从海报或者展览信息电子标签上读取相关信息。

3. NFC 与蓝牙的关系

NFC 和蓝牙都是短程通信技术，而且都被集成到移动电话。但 NFC 不需要复杂的设置程序。NFC 也可以简化蓝牙连接。

NFC 略胜蓝牙的地方在于设置程序较短，但无法达到蓝牙的低功率。蓝牙由于其配对过程相对较长，因此不太适用于类似刷卡的即时性动作。

NFC 的最大数据传输量是 424kbps，远小于 Bluetooth V2.1(2.1 Mbps)。虽然 NFC 在传输速度与距离比不上蓝牙，但是 NFC 技术不需要电源，对于移动电话或是行动消费性电子产品来说，NFC 使用比较方便。NFC 的短距离通信特性正是其优点，由于耗电量低、一次只和一台机器连接，所以拥有较高的保密性与安全性，NFC 有利于信用卡交易时避免被盗用。NFC 的目标并非是取代蓝牙等其他无线技术，而是在不同的场合、不同的领域起到相互补充的作用。

8.1.6 小结

前面所介绍的几种网络通信技术，它们之间并不是相互冲突的关系，而是各自有各自的适用领域，它们在各自的适用领域中都能够发挥出比其他技术更加优越的水平，因此，需要

分清楚这些技术之间的关系。在实际的应用中,根据它们的优缺点来选择合适的技术,这样才能够达到事半功倍的效果。

8.2 Http 通信

在 8.1 节中已经提到,Android 平台提供了 3 种网络接口,分别是 java. net. * 、org. apache. * 和 android. net. * 。可以使用这些接口方便地进行 Android 网络编程。接下来将依次使用这几种接口来进行网络开发,本节首先介绍与 Http 相关的接口。在网络应用中,通过访问 url 来获取 Web 页面是常见的获取信息方式,为此,Android 提供了 HttpClient 和 HttpURLConnection 接口来开发 Http 程序。

8.2.1 Http 简介

Http(Hypertext Transfer Protocol)即超文本传送协议,它是 Web 的基础协议,是建立在 TCP 上的一种应用。Http 连接最显著的特点就是客户端发送的每次请求都需要服务器返回响应,并在请求结束后释放连接,这个建立连接到关闭连接的过程称为“一次连接”。由于 HTTP 在每次请求结束后都会主动释放连接,因此 HTTP 连接是一种“短连接”、“无状态”的连接。在 Http 1.0 时期,要保持客户端程序的在线状态,需要不断地向服务器发送连接请求。通常的做法是即使不需要请求任何数据,客户端也保持每隔一段固定的时间向服务器发送一次“保持连接”的请求,服务器在收到该请求后对客户端进行回复,表明知道客户端“在线”。若服务器长时间无法收到客户端的请求,则认为客户端“下线”,若客户端长时间无法收到服务器的回复,则认为网络已经断开。而在 Http 1.1 版本时增加了持久连接支持,即是将关闭连接的主动权交给客户端,只要客户端没有请求关闭连接,就可以持续向服务器发送 Http 请求。HTTP 1.1 除了支持持久连接外,还将 HTTP 1.0 的请求方法从原来的 3 个(GET、POST 和 HEAD)扩展到了 8 个(OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE 和 CONNECT);而且还增加了很多请求和响应字段,如持久连接的字段 Connection。这个字段有两个值:Close 和 Keep-Alive。如果使用 Connection:Close,则关闭 HTTP 1.1 的持久连接的功能。若要打开 HTTP 1.1 的持久连接的功能,必须将字段设置为 Connection:Keep-Alive,或者不加 Connection 字段(因为 HTTP 1.1 在默认情况下就是持久连接的)。另外,还提供了身份认证、状态管理和缓存(Cache)等相关的请求头和响应头。总结起来 Http 主要有如下几个特点:

(1) 支持客户/服务器模式。

(2) 简单快速——客户向服务器请求服务时,只需传送请求方法和路径。请求方法常用的有 GET、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单,使得 HTTP 服务器的程序规模小,因而通信速度很快。

(3) 灵活——HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

(4) 无状态——HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息,则它必须重传,这样可能导致每次连接

传送的数据量增大。另外，在服务器不需要先前信息时它的应答就较快。

8.2.2 使用 HttpClient 接口

Http 请求数据通常使用 GET 和 POST 向服务器提交表单而获取响应的方式获得数据，表单提交的 GET 和 POST 方法都可以向服务器请求数据，它们主要有以下几点区别：

(1) GET 方法是将参数数据队列附加到 URL 的 ACTION 属性中，值和表单内各个字段一一对应，以明文的方式存在于 URL 中。而 POST 方法则是将数值内容放置在 HTML HEADER 内一起传送至 ACTION 属性所指的 URL 地址，这个过程对于用户来说是不可见的。

(2) 对于 GET 方法，服务端采用 Request.QueryString 获取变量的值，而对于 POST 方法，服务端采用 Request.Form 获取提交的数据。

(3) 一般来说，GET 方法向服务器传送的数据量较小，不能大于 2KB(URL 长度限制)。而 POST 方法传送的数据量较大，一般认为是不受限制。但在实际应用上，IIS4 中最大数据量为 80KB，而 IIS5 中为 100KB。

使用 Apache 提供的 HttpClient 接口可以方便地进行 Http 操作，对于 GET 方法和 POST 方法的使用有所不同，通过下面的示例(本节配套示例为 HttpClientDemo)来进行说明，使用 GET 方法请求数据的代码如下：

```
protected void httpClientGet() {
    //GET 请求的 url,可以看到 url 中 weather 的值为 chengdu
    String googleWeatherUrl = "http://www.google.com/ig/api?hl=zh-cn&weather=chengdu";
    DefaultHttpClient httpclient = new DefaultHttpClient();
    HttpGet httpget = new HttpGet(googleWeatherUrl);
    //ResponseHandler,用于处理服务端返回的响应
    ResponseHandler<String> responseHandler = new BasicResponseHandler();

    try {
        String content = httpclient.execute(httpget, responseHandler);
        Toast.makeText(getApplicationContext(), "连接成功!",
                Toast.LENGTH_SHORT).show();
        //设置 TextView,显示获取的网页内容
        httpContent.setText(content);
    } catch (Exception e) {
        Toast.makeText(getApplicationContext(), "连接失败", Toast.LENGTH_SHORT)
        .show();
        e.printStackTrace();
    }
    httpclient.getConnectionManager().shutdown();
}
```

在使用 Get 方法请求数据的方式下，主要使用到了几个类，它们来自于 apache 提供的 API：

```
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.impl.client.BasicResponseHandler;
```

可以发现,Get 请求是由 HttpClient 对象的 execute 方法发出的,execute 方法原型为:

```
public <T> T execute( final HttpUriRequest request, final ResponseHandler<? extends T>
responseHandler)
    throws IOException, ClientProtocolException
```

参数列表:

- request——该对象包含要请求的 Uri 信息,在示例中即为一个 HttpGet 对象,HttpGet 对象实现了 HttpUriRequest 接口。
- responseHandler——处理服务器响应的对象,在示例中直接使用了一个简单的对象 BasicResponseHandler 来处理服务器返回的响应,该 Handler 就是简单地将服务器响应作为字符串直接返回,因此该方法的返回值也是 String 类型。

返回值:

- T——该泛型对象由 responseHandler 决定,此处将返回一个 String 对象。

于是,当执行第 10 行代码时,如果网络连接正常,content 值将会是服务器所返回的响应,由于代码中使用的 URL 是 Google 提供的一个简易的天气服务,所以返回的是一个 xml 文档,content 的值就是未经解析的 xml 文档完整内容,如图 8-7 所示。



图 8-7 通过 HttpGet 方法取得的内容

可以看到,返回的内容就是代表当前天气的 xml 文档,解析该 xml 文档的方法将在 8.4 节的第 1 部分"通过 Http Get 方式"中进行说明。

下面再来看一看如何使用 POST 方法请求数据,代码如下:

```
protected void httpClientPost()
{
    // 定义需要获取的内容来源地址
    String SERVER_URL = "http://webservice.webxml.com.cn/" +
            "WebServices/WeatherWS.asmx/getWeather";
    // 创建 HttpPost 请求
    HttpPost request = new HttpPost(SERVER_URL);
    // 添加参数
    List<BasicNameValuePair> params = new ArrayList<BasicNameValuePair>();
    params.add(new BasicNameValuePair("theCityCode", "成都"));
    params.add(new BasicNameValuePair("theUserID", ""));
```

```
12        try
13        {
14            //设置参数的编码
15            request.setEntity(new UrlEncodedFormEntity(params, HTTP.UTF_8));
16            //发送请求并获取反馈,解析返回的内容
17            HttpResponse httpResponse = new DefaultHttpClient().execute(request);
18            //如果返回状态码不为404,则显示获取的内容
19            if (httpResponse.getStatusLine().getStatusCode() != 404)
20            {
21                String result = EntityUtils.toString(httpResponse.getEntity());
22                httpContent.setText(result.toString());
23            }
24        }catch (Exception e) {}
25    }
```

在使用 Post 方法请求数据的方式下，主要使用到了如下几个类，它们同样来自于 apache 提供的 API：

```
import org.apache.http.HttpResponse;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.protocol.HTTP;
import org.apache.http.util.EntityUtils;
```

在使用 POST 方法请求数据时，也可以使用与前面 GET 方法同样的方式，使用前面提到的那个 HttpClient.execute()方法，然后返回一个代表网页数据的字符串。不过此处为了不重复而采用了另一种方式，同样是使用 execute()方法，但是这里 execute()方法仅有一个参数，而不再传入 responseHandler 对象，execute()方法原型为：

```
public final HttpResponse execute (HttpUriRequest request) throws IOException,
ClientProtocolException
```

参数列表：

- request——该对象包含要请求的 Uri 信息，在示例中即为一个 HttpPost 对象，HttpPost 对象实现了 HttpUriRequest 接口。

返回值：

- HttpResponse——该对象封装了服务器返回的数据。

不同于前面所使用的方式，这里通过 execute 方法获取了封装有服务器返回的数据的 HttpResponse 对象(代码第 17 行)，然后再使用 HttpResponse 的相关方法来获取具体的数据(代码第 21 行)，即使用 httpResponse.getEntity()方法获取一个 HttpEntity 对象并将其解析为字符串，从而达到了同样的效果。另外可以注意到代码第 19 行通过一系列的方法获取了服务器响应的状态码，并以此来判别访问的状态。使用 POST 方法获取的数据如图 8-8 所示，其效果与图 8-7 相似，读者可以根据两种方法的代码来体会这两种方式的不同。



图 8-8 通过 HttpPost 方法取得的数据

8.2.3 使用 HttpURLConnection 接口

使用 HttpClient 可以快速开发出功能强大的 Http 程序。不过一般来说，要开发与 Internet 连接的程序，最基础的还是使用 HttpURLConnection。使用 HttpURLConnection 提供的方法，可以方便地对网络资源进行访问，读者可以通过查询 Android API 文档来了解其提供的所有方法。

HttpURLConnection 是 Java 提供的网络访问接口，它继承自 URLConnection。要获取 HttpURLConnection 类的实例，需要使用 openConnection()方法来获取，代码如下：

```
String Url = "http://www.google.com/ig/api?weather=chengdu";
URL url = new URL(googleWeatherUrl);
HttpURLConnection httpConn = (HttpURLConnection) url.openConnection();
```

获取了 HttpURLConnection 实例后，需要使用它的 getInputStream()获取输入流，然后从该输入流中获得数据，读输入流完成后关闭输入流并断开连接即可，由于 HttpURLConnection 默认使用 GET 方法获取数据，因此不需要在代码中进行说明，在使用 POST 方法时需要使用 setRequestMethod()将访问方法设置为 POST。还是通过示例来进行说明，本节的配套示例为 HttpURLConnectionDemo。HttpURLConnection 使用 GET 方法获取数据的代码如下：

```
//使用 URLConnection, GET 方式连接 GoogleWeatherAPI
protected void urlGetConn() {
    String googleWeatherUrl = "http://www.google.com/ig/api?weather=chengdu";
```

```
04        try {
05            URL url = new URL(googleWeatherUrl);
06            //获取 HttpURLConnection 实例
07            HttpURLConnection httpConn = (HttpURLConnection) url.openConnection();
08            if (httpConn.getResponseCode() == HttpURLConnection.HTTP_OK) {
09                Toast.makeText(getApplicationContext(), "连接成功",
10                        Toast.LENGTH_SHORT).show();
11                //InputStreamReader 用于读取网页内容
12                InputStreamReader isr = new InputStreamReader(httpConn.getInputStream(),
13                    "utf-8");
14                int i;
15                String content = "";
16                //读取数据
17                while ((i = isr.read()) != -1) {
18                    content = content + (char) i;
19                }
20                isr.close();
21                tv.setText(content);
22            }
23            httpConn.disconnect();
24        } catch (Exception e) {
25            Toast.makeText(getApplicationContext(), "连接失败",
26                 Toast.LENGTH_SHORT).show();
27            e.printStackTrace();
28        }
29    }
```

对比前面使用 HttpClient 的代码，可以发现使用 HttpURLConnection 的方式非常简捷，HttpURLConnection 可以通过 getInputStream()方法直接获取到指定 Url 的输入流（第 12 行），然后就可以直接根据这个流来获取 Url 的数据（第 12～19行）。最后，服务器返回的信息全部被存放在 content 中，显示界面如图 8-9 所示。

HttpURLConnection 使用 POST 方法请求数据需要改变的主要有两处：一就是需要使用 setRequestMethod()方法将请求方法设置为“POST”；二是使用 DataOutputStream 向服务器写入参数值，例如指定网站的登录用户名和密码，以手机人人网的登录为例，代码如下：

```
public void urlPostConn(){
     String httpUrl = "http://3g.renren.com/login.do";
    String resultData = "";
    URL url = null;
    try {
        url = new URL(httpUrl);
    } catch (MalformedURLException e) {
```



图 8-9　HttpURLConnection GET 方式获取数据

```
08                 e.printStackTrace();
09             }
10             if(url != null){
11                 try {
12                     HttpURLConnection urlConn = (HttpURLConnection)url.openConnection();
13 
14                     //使用 Post 方式的相关设置
15                     urlConn.setDoOutput(true);
16                     urlConn.setDoInput(true);
17                     urlConn.setRequestMethod("POST");
18                     urlConn.setUseCaches(false);
19                     urlConn.setInstanceFollowRedirects(true);
20                     urlConn.setRequestProperty("Content-Type",
21                        "application/x-www-form-urlencoded");
22                     urlConn.connect();
23                     //上传参数,此处的 content 中相应部分填入你拥有的账户和密码
24                     DataOutputStream out = new DataOutputStream(urlConn.getOutputStream());
25                     String content = "email=youremail@mail.com&password=yourpassword";
26                     out.writeBytes(content);
27 
28                     out.flush();
29                     out.close();
30 
31                     //读取数据
32                     InputStreamReader in = new InputStreamReader(urlConn.getInputStream());
33                     BufferedReader buffer = new BufferedReader(in);
34                     String str = null;
35                     while((str = buffer.readLine()) != null){
36                         resultData += str + "\n";
37                     }
38                     in.close();
39                     urlConn.disconnect();
40                     tv.setText(resultData);
41                 } catch (IOException e) {
42                     e.printStackTrace();
43                 }
44             }
45             else {
46                 tv.setText("URL null");
47             }
48     }
```

如上面的代码所示,在使用 openConnection()方法获取了 HttpURLConnection 对象之后,首先进行了一系列的设置,这些设置为使用 POST 方法传送数据作准备(代码第 15～10 行),然后使用了 DataOutputStream 向服务器传送了参数(代码第 24～26 行),之后就与前面的 GET 方法一样使用 InputStreamReader 读取服务器返回的信息。注意代码中第 25 行 content 的值需要改为已知的一对正确的用户名和密码,否则将会出现密码错误之类的提示,这些提示信息也可以从返回的数据中看到,如果用户名密码正确,应该能够在返回的信息中发现一些熟悉的数据(例如自己的人人网新鲜事、自己的状态签名等)。其运行结果如图 8-10 所示。

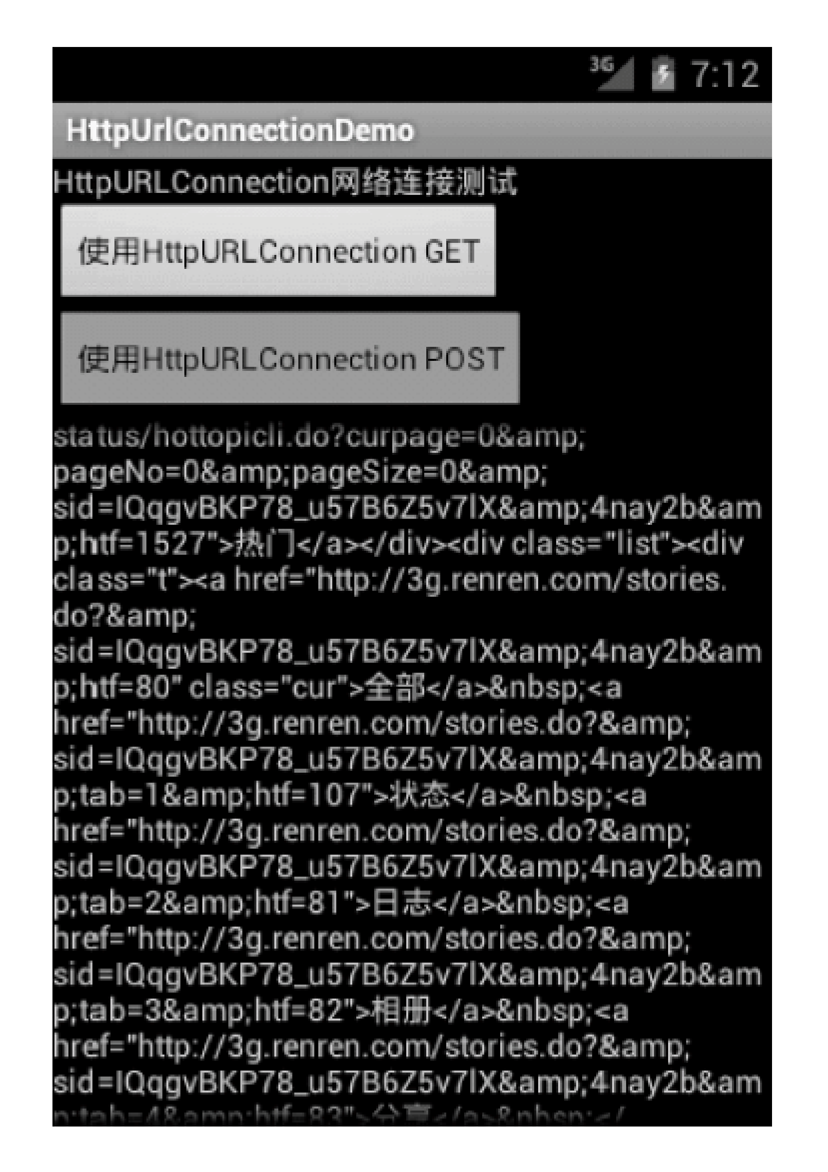


图 8-10 HttpURLConnection POST 方式

8.3 Socket 通信

8.3.1 Socket 简介

本节所要介绍的 Socket 是针对于网络范畴的 Socket，Socket 的本身含义为“插座”，这里通常译为“套接字”，这个翻译比较形象地说明了 Socket 的含义：利用 Socket 可以使得两个通信末端通过“套接”的方式来建立其连接。Socket 用于描述 IP 地址和端口，是一个通信链的句柄。例如，在 Internet 上的主机一般运行了多个服务软件，同时提供多种服务，在这样的条件下，每一个服务都会建立一个 Socket，并绑定到一个端口，从而使不同的端口对应着不同的服务。Socket 又正如其英文原意那样，就像一个“多孔插座”：一台主机犹如布满各种插座的房间，每个插座有一个编号，有的插座提供 220 伏交流电，有的提供 110 伏交流电，有的提供有线电视节目等。客户软件只需要将插头插到不同编号的插座，就可以得到不同的服务。在计算机网络中，Socket 用于连接网络中的某一对基于互联网协议的计算机网络中的不同进程，使得这些进程可以跨越网络进行通信。Socket 通常也被看作是 TCP/IP 协议栈所提供的 API，这些 API 通常由操作系统提供。Socket 建立了一套用于将由外界发送过来的数据包转发至合适的应用程序进程或线程的机制，这种机制基于一系列本地和远程的 IP 地址和端口号。每一个 Socket 端口到应用程序的映射都由操作系统来负责完成。一个完整的 Socket 地址由 IP 地址和端口号组成，就好比一个完整的电话号码由区号和号码组成。以一个国家级电话网为例，电话的通话双方相当于相互通信的 2 个进程，区号是它的网络地址；区内一个单位的交换机相当于一台主机，主机分配给每个用户的局内号码相当于 Socket 号。任何用户在通话之前，首先要占有一部电话机，相当于申请一个 Socket；其次要知道对方的号码，相当于对方有一个固定的 Socket。然后向对方拨号呼叫，相当于

发出连接请求(如果对方不在同一区内,则需要加拨对方区号,相当于给出网络地址)。对方假如在场并空闲(相当于通信的另一主机开机且可以接受连接请求),于是在一方呼叫后另一方拿起电话话筒,双方就可以正式通话,即连接成功。双方通话的过程,是一方向电话机发出信号和对方从电话机接收信号的过程,相当于向 Socket 发送数据和从 Socket 接收数据。通话结束后,一方挂掉电话机就相当于关闭了这条连接。

一条完整的 Socket 连接由如下几个部分组成:

- 本地 Socket 地址——包括本地 IP 和端口号。
- 远程 Socket 地址——这个部分仅当 Socket 连接被建立时才存在,通常一个服务端的 Socket 可以同时服务多个客户端 Socket。
- 协议:该连接上传输数据所遵循的协议,例如 TCP、UDP、Raw IP 等,使用这些不同的传输协议传输数据的 Socket 在实现上是完全不同的。

由于一条完整的 Socket 连接由上述 3 个部分组成,因此其中任何一个条件不同都会得到一个完全不同的 Socket 连接,因此,一个服务端可以在一个端口号上面同时与多个客户端进行通信而互不影响,因为这些 Socket 连接都是独立存在的。

套接字是通信的基础,一个套接字就代表了通信的一端。一个正在被使用的套接字都有着与其相关的进程。套接字存在于通信域中,通信域是为了处理一般的线程通过套接字通信而引进的一种抽象概念。套接字通常和同一个域中的套接字交换数据(数据交换也可能穿越域的界限,但这时一定要执行某种解释程序)。例如,Windows Sockets 规范支持单一的通信域,即 Internet 域。各种进程使用这个域互相之间用 Internet 协议族来进行通信(Windows Sockets 1.1 以上的版本也支持其他的域,例如 Windows Sockets 2)。应用程序一般仅在同一类型的套接字之间进行通信。不过只要底层的通信协议允许,不同类型的套接口间也照样可以通信。

8.3.2 Socket 类型

常用的 Socket 类型有两种,即流式 Socket 和数据报式 Socket;还有一种不常用的类型即原始 Socket(Raw IP socket),它们的含义分别为:

- 流式 Socket——即通常意义上的“面向连接的”Socket,这种 Socket 使用 TCP 协议或者 SCTP(Stream Control Transmission Protocol)协议,针对于面向连接的 TCP 服务应用。
- 数据报式 Socket——数据报式 Socket 是一种无连接的 Socket,针对于无连接的 UDP 服务应用,这种 Socket 使用 UDP 协议。
- Raw IP Socket——这种 Socket 通常存在于路由器或者其他类似的网络设备中,它们绕过传输层,并且数据报的头部不会被剥离,但是却能够被引用程序访问。

8.3.3 Socket 连接过程

处于网络中不同主机中的进程之间要进行通信,首先必须解决的是如何唯一地标识一个进程。众所周知,在本地计算机中可以通过进程标识符 PID 来唯一地标识一个进程,但是在拥有多台计算机的网络中,这种标识是行不通的。8.3.1 节已经提到,一条完整的

Socket 连接由 3 个部分构成，其中包括了 IP 地址、端口号和协议。TCP/IP 协议族(如图 8-11 所示)解决了这个问题，因为网络层的"IP 地址"可以唯一标识网络中的主机，而传输层的"协议＋端口号"可以唯一地标识主机中的应用程序(进程)。利用这个三元组(IP 地址，协议，端口)就可以唯一地标识网络上的进程，网络中的进程通信就是利用这个标志与其他进程进行交互的。就目前而言，几乎所有的应用程序都采用 Socket 通信，而网络中的进程通信又是无处不在的，因此在网络世界中，可以毫不夸张地说："一切皆 Socket"。

应用层
传输层(协议＋端口号)
网络层(IP 地址)
链路层

图 8-11 TCP/IP 四层结构

由于基于 UDP 协议的 Socket 连接发送消息的过程比较简单，因此下面仅对基于 TCP 协议的 Socket 连接的建立过程进行说明，并且对服务端和客户端分别进行说明，由于 Android 的应用层采用的是 Java 语言，所以 Java 支持的网络编程方式 Android 都能够很好地支持，因此下面站在 Java 的角度来进行说明。

服务器端在建立一个 Socket 连接时的步骤如下。

(1) 在使用 Socket 之前，要初始化 Socket，在 Java 中就是新建一个 ServerSocket 对象，ServerSocket 的构造方法参数通常为一个端口数值。

(2) 在初始化正常完成以后(端口未被占用)，就建立了服务端的 Socket，然后可以通过 ServerSocket.accept 方法开始侦听网络中的连接请求，在客户端连接到服务端之前，ServerSocket 所处的线程一直处于阻塞状态。

(3) 当检测到来自客户端的连接请求时，服务端会向客户端发送收到连接请求的信息，从而建立起与客户端之间的连接，ServerSocket 的 accept 方法将会返回一个代表了连接到远程客户端的 Socket，并且停止阻塞，线程继续执行。

(4) 在通信的过程中，服务器端通过远程 Socket 获取到数据输出流 DataOutputStream，从而通过这个输出流向客户端发送数据。

(5) 当完成通信后，服务端关闭与客户端的 Socket 连接。

客户器端在建立一个 Socket 连接时的步骤如下。

(1) 初始化并建立客户端的 Socket，在构造方法中确定需要连接到的服务器的主机名/IP 和端口。

(2) 发送连接请求到服务器，在 Java 中，建立客户端 Socket 的同时将会发送连接请求，然后等待服务器的回馈信息。

(3) 连接成功后，可以使用 Socket 的 getInputStream()和 getOutputStream()方法获取到来自服务器的数据输入流和输出流，从而与服务器进行数据交互。

(4) 数据处理完毕后，关闭自身的 Socket 连接。

客户端和服务端的 TCP Socket 通信过程的流程图如图 8-12 所示。

8.3.4 Socket 通信示例

正如 8.1 节所述：Android 的应用层采用的是 Java 语言，所以 Java 支持的网络编程方式 Android 都能够很好地支持。本节就将在 Android 模拟器上实现一个建议的 Socket 通信示例，从示例中读者可以发现，利用 Java 的 Socket 通信相关 API，相关功能的实现在 Android 平台上也十分简便。

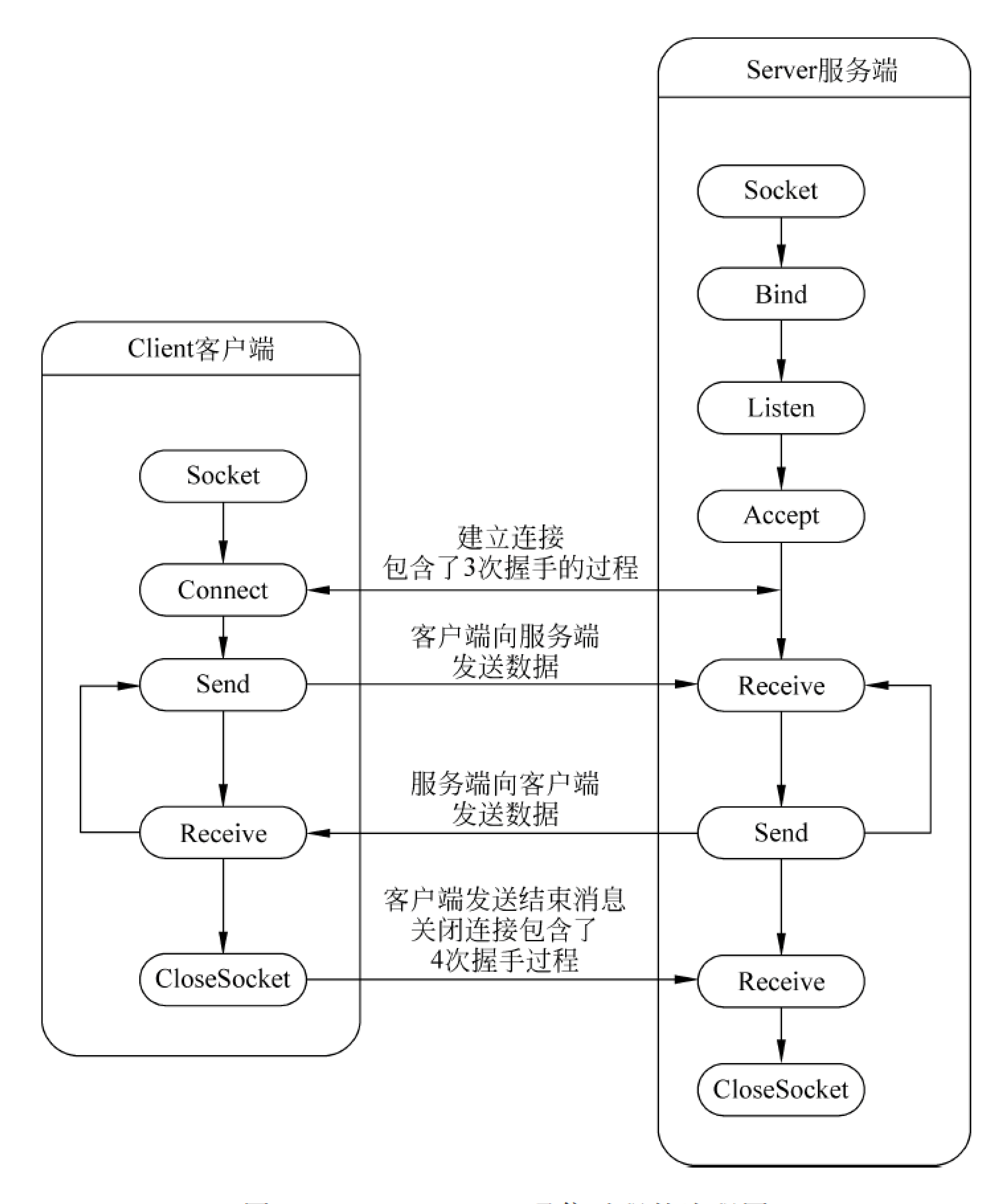


图 8-12 TCP Socket 通信过程的流程图

1. 示例功能说明

配套示例的名称为 TestSocketClient 和 TestSocketServer。本示例要实现的功能是：分别实现服务端和客户端的 Android 应用，然后在两端之间建立 Socket 连接，并且由服务端向客户端传送一个指定的文件，要求在传送文件之前服务器先将待传送文件的文件名和文件大小发送给客户端，并且在两端的界面上显示文件传输的进度，具体的效果如图 8-13～图 8-17 所示。

如图 8-13～图 8-17 所示，服务端和客户端是分别运行在两台模拟器上的，首先在一台模拟器上运行服务端程序，如果 Socket 成功绑定端口，则界面中将出现如图 8-12 所示的提示"正在等待连接，监听端口：55555"，这个 55555 的端口号是在代码中指定的；然后再在另一台模拟器上启动客户端程序，启动之后将成功连接至服务端然后双方建立起 Socket 连接，服务端开始向客户端传递文件，如图 8-13 和图 8-14 所示；经过一段时间的传输之后，服务端发送文件完成（如图 8-15 所示），客户端接收完成并成功保存文件（如图 8-16 所示）。由于是在模拟器上面对示例进行测试，因此模拟器之间并不能够直接建立 Socket 连接，而是需要通过宿主 PC 主机的转发，具体的操作将在随后进行说明。




图 8-13　服务端启动

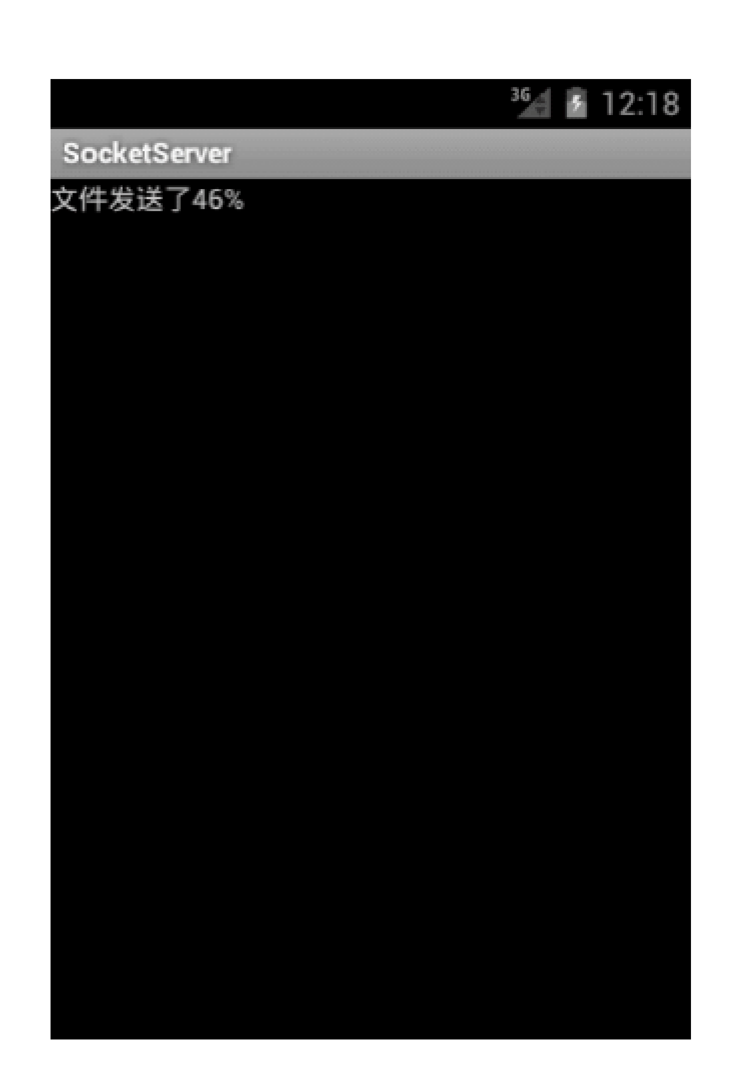


图 8-14　客户端连接后服务端发送文件

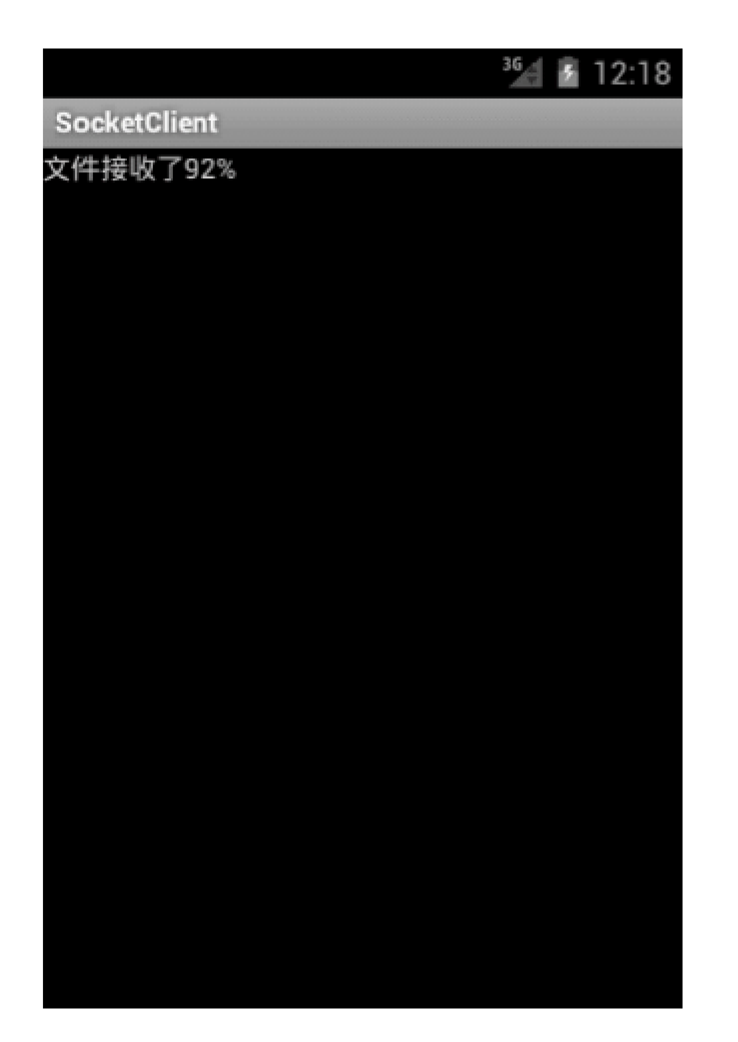


图 8-15　客户端接收文件

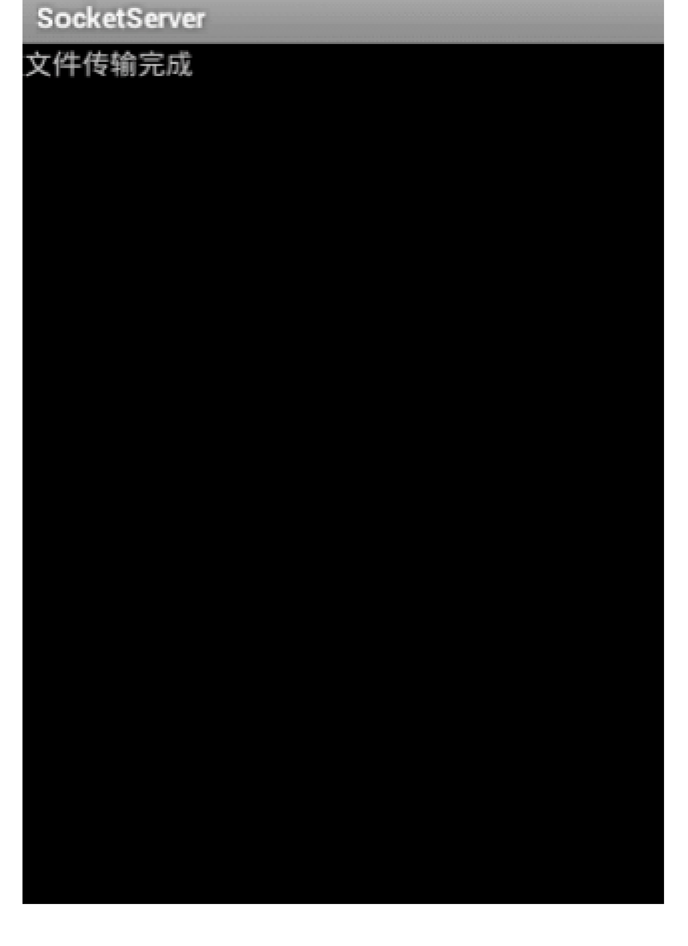


图 8-16　服务端发送完成

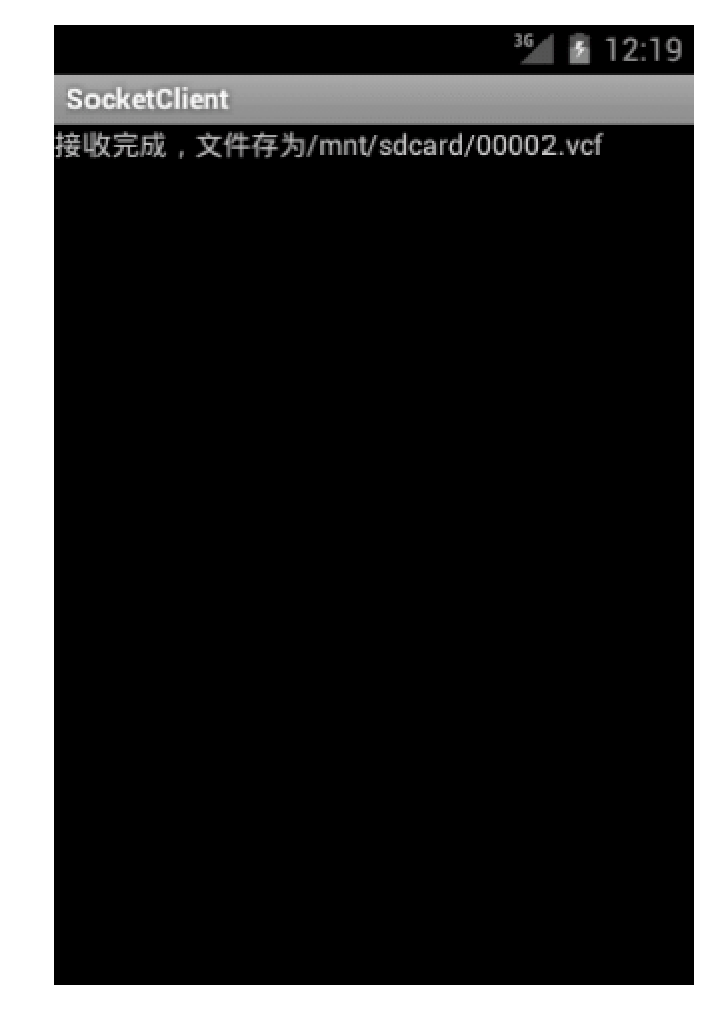


图 8-17　客户端接收完成

2. 服务端实现

服务端主要由一个监听线程实现，该监听线程初始化 ServerSocket 后，调用其 accept 方法等待客户端的连接，在客户端连接成功后进行数据的传输，该监听线程的代码如下：

```
class ServerThread extends Thread {
    @Override
    public void run() {
        try {
            ServerSocket ss = new ServerSocket(SERVERPORT);
            updateDebug("正在等待连接,监听端口：" + ss.getLocalPort());
            while (true) {
                s = ss.accept();
```

```
09                  updateDebug("已连接客户端");
10                  DataInputStream dis = new DataInputStream(new BufferedInputStream
11                      (s.getInputStream()));
12                  //首先从客户端读取一段数据,确认已正确连接
13                  dis.readByte();
14
15                  //待发送的文件
16                  String filePath = "/mnt/sdcard/00002.vcf";
17                  File fi = new File(filePath);
18
19                  //发送文件名和文件长度信息
20                  DataOutputStream fileOutput =
21                      new DataOutputStream(s.getOutputStream());
22                  fileOutput.writeUTF(fi.getName());
23                  fileOutput.flush();
24                  long len = fi.length();
25                  fileOutput.writeLong(len);
26                  fileOutput.flush();
27                  //发送文件
28                  DataInputStream fileInput = new DataInputStream
29                      (new BufferedInputStream(new FileInputStream(filePath)));
30                  int bufferSize = 8192;
31                  byte[] buf = new byte[bufferSize];
32                  int passedlen = 0;
33                  while (true) {
34                      int read = 0;
35                      if (fileInput != null) {
36                          read = fileInput.read(buf);
37                      }
38                      if (read == -1) {
39                          break;
40                      }
41                      fileOutput.write(buf, 0, read);
42                      passedlen += read;
43                      updateDebug("文件发送了" + (passedlen * 100/ len) + "%\n");
44                  }
45                  fileOutput.flush();
46                  fileInput.close();
47                  s.close();
48                  updateDebug("文件传输完成");
49              }
50              }catch(Exception e) {
51                  e.printStackTrace();
52          }
53      }
54  }
```

其中,代码第 05 行建立了用于监听的 ServerSocket 端口,然后线程开始在一个 while(true) 循环体中监听客户端的连接请求(代码第 07~49 行);当没有客户端连接时,循环体阻塞在 accept 方法处(代码第 08 行),一旦有来自客户端的连接请求,accept 方法将被执行,同时为服务端返回一个用于与客户端通信的 Socket 对象;示例在 Socket 连接建立成功之后

并没有直接开始用于传输文件相关的操作，而是首先获取来自客户端的输入流（代码第 10 行和第 11 行），并且等待由客户端发送过来的一段确认数据（代码第 13 行），同样，当客户端没有向服务端发送数据时，代码将会阻塞在 readByte 方法处；一旦接收到了来自客户端发送的信息，代码将继续向下执行，在第 16 行和第 17 行确定要进行发送的文件，然后获取向客户端发送数据的输出流（代码第 20 行）并向客户端发送文件名及文件长度信息（代码第 21～25 行），其中文件名用于客户端为接收到的文件命名，文件长度用于客户端确认收到的文件的完整性，并且为传输进度的显示提供依据；代码第 28～49 行则是服务端具体的发送文件的代码，服务器从文件输入流获取数据（代码第 28～31 行、第 34～40 行代码）并通过输出流向客户端发送文件（代码第 41 行），passedlen 变量（代码第 32 行）用于保存当前已发送的文件长度，发送进度就是通过该变量的值与文件总长度（代码第 23 行）的比值来计算的。

服务端要开始监听线程的执行，只需要在 Activity 的 onCreate 方法中创建一个 ServerThread 并且执行其 start 方法即可。

3. 客户端实现

本小节将介绍客户端的具体实现，为了使代码更加清晰，客户端的实现包括了两个类：

- ClientSocket——该类对 Socket 的操作进行了封装，提供了用于建立连接的 CreateConnection 方法、发送消息的 sendMessage 方法，获取输入流的 getMessageStream 方法以及关闭 Socket 连接的 shutdownConnection 方法。
- client——客户端 Activity，使用 ClientSocket 来建立连接，并且实现文件的接收功能。

首先来看一下 ClientSocket 的实现，分别简要地介绍其提供的几种方法。

1）ClientSocket 构造方法

提供了一个构造方法，该构造方法包括两个参数，即需要连接到的 IP 和端口地址：

```
public ClientSocket(String ip, int port) {
    this.ip = ip;
    this.port = port;
}
```

2）void CreateConnection()

该方法用于创建 Socket 连接，实际上就是构造一个 Socket 对象并捕获异常：

//创建 socket 连接

```
public void CreateConnection() throws Exception {
    try {
        socket = new Socket(ip, port);
    } catch (Exception e) {
        e.printStackTrace();
        if (socket != null) socket.close();
    }
}
```

3）void sendMessage()

该方法用于向服务端 Socket 发送一条确认消息，用于确认两端之间的 Socket 连接已经正确地建立起来：

```
public void sendMessage() throws Exception {
    try {
        out = new DataOutputStream(socket.getOutputStream());
        out.writeByte(0x1);
        out.flush();
        return;
    } catch (Exception e) {
        e.printStackTrace();
        if (out != null) out.close();
    }
}
```

4）DataInputStream getFileInputStream()

该方法用于从建立好的 Socket 连接处获取输入流，从而使得调用方能够从该输入流中读取信息：

```
public DataInputStream getFileInputStream() throws Exception {
    try {
        getFileInputStream = new DataInputStream(new BufferedInputStream
            (socket.getInputStream()));
        return getFileInputStream;
    } catch (Exception e) {
        e.printStackTrace();
        if (getFileInputStream != null)
            getFileInputStream.close();
        throw e;
    }
}
```

5）void shutdownConnection()

该方法负责在 Socket 连接不再被使用时释放相关资源并关闭连接：

```
public void shutdownConnection() {
    try {
        if (out != null)
            out.close();
        if (getMessageStream != null)
            getMessageStream.close();
        if (socket != null)
            socket.close();
    } catch (Exception e) {
    }
}
```

实现了 ClientSocket 类之后，再来实现客户端的 Activity(client 类)，该 Activity 的作用即是根据服务器的 IP 和端口信息创建一个连接到服务器的 Socket 连接，然后发送一条确

认消息与服务端进行握手，握手成功后再通过一个接收线程来接收由服务端发送过来的文件并保存在指定的文件夹，同时在 Activity 界面上更新文件传输的状态信息。有了 ClientSocket 类的实现，在 client 中只需要直接调用 ClientSocket 的相关方法即可，client 中主要包括了如下几个方法和线程类：

- createConnection()——创建连接(向下调用 ClientSocket 类的 CreateConnection 方法)。
- sendMessage()——发送一条消息(向下调用 ClientSocket 类的 sendMessage 方法)。
- GetFileThread()——接收文件的线程，在线程的 run()中调用 getMessage 方法，以免阻塞用户界面。
- getFile()——从 Socket 中接收由服务器发送过来的文件，并做相应的状态更新和文件保存工作。

其中前面两个方法的内容十分简单，这里就不给出代码了，接收文件的线程及 getFile 方法的代码如下：

```
private class GetFileThread extends Thread{

    @Override
    public void run() {
        getFile();
    }
}

private void getFile() {
    if (cs == null)
        return;
    DataInputStream inputStream = null;
    try {
        inputStream = cs.getFileInputStream();
    } catch (Exception e) {
        updateDebug("接收文件错误\n");
        return;
    }
    try {
        String savePath = "/mnt/sdcard/";//文件保存路径

        //获取文件名和文件长度信息
        long len = 0;
        savePath += inputStream.readUTF();
        len = inputStream.readLong();
        updateDebug("文件的长度为:" + len + "\n");
        updateDebug("开始接收文件!" + "\n");

        //接收文件
        int bufferSize = 8192;
        byte[] buf = new byte[bufferSize];
        int passedlen = 0;
        DataOutputStream fileOut = new DataOutputStream(new BufferedOutputStream
                (new BufferedOutputStream(new FileOutputStream(savePath))));
        while (true) {
            int read = 0;
```

```
37                 if (inputStream != null) {
38                     read = inputStream.read(buf);
39                 }
40                 passedlen += read;
41                 if (read == -1) {
42                     break;
43                 }
44                 updateDebug("文件接收了" +  (passedlen * 100/ len) + "%\n");
45                 fileOut.write(buf, 0, read);
46             }
47             updateDebug("接收完成,文件存为" + savePath + "\n");
48             fileOut.close();
49             cs.shutdownConnection();
50         } catch (Exception e) {
51             updateDebug("接收文件错误" + "\n");
52             return;
53         }
54     }
```

getFile 方法与前面服务端的 ServerThread 线程的循环体结构是一一对应的关系：在代码第 14 行得到了文件的输入流之后，第 24 行的 inputStream.readUTF()方法将从输入流中读取一个以 UTF-8 编码的字符串(即文件名)，该方法对应于 ServerThread 代码的第 21 行 fileOutput.writeUTF(fi.getName())；第 25 行的 inputStream.readLong()方法将从输入流中读取一个 64 位的 Long 型数据(即文件长度)，对应于 ServerThread 代码的第 23 行和第 24 行；之后便是接收文件的代码(代码第 29～49 行)，同样对应于 ServerThread 发送文件的代码。正是通过这样的方式实现了服务端到客户端的正确通信。

8.4 Web 服务

8.4.1 Web 服务简介

Web Service(Web 服务)是一种基于 XML 和 HTTP 技术的服务，它部署在 Web 服务器上，由 Web 服务器管理。Web 服务是一种面向服务的架构的技术，通过标准的 Web 协议提供服务，目的是保证不同平台的应用服务可以互操作。通过 Web 服务，它使得不同计算机语言、不同计算机平台之间的方法调用成为可能，是远程调用和分布式系统的重要实现手段。

从表面上看，Web 服务就是一个应用程序，它向外界暴露出一个能够通过 Web 进行调用的 API。这就是说，能够用编程的方法通过 Web 调用来实现某个功能的应用程序。例如最常用的天气查询 Web 服务，它的作用是查询天气预报信息。它接受某个地点的标志(城市名或邮编)作为查询字符串，返回该地点的天气预报具体信息。也可以在浏览器的地址栏中直接输入 HTTP GET 请求来获取天气预报，例如查询成都的天气预报可以通过 http://www.google.com/ig/api?hl=zh-cn&weather=chengdu 来得到，读者可以在浏览器中输入该 Url 来体验一下 Web 服务。

从深层次上看，Web Service 是一种新的 Web 应用程序分支，它们是自包含、自描述、模块化的应用，可以在网络（通常为 Web）中被描述、发布、查找以及通过 Web 来调用。

根据 W3C 的定义，Web 服务应当是一个软件系统，它用于支持网络间不同机器的互操作，因此，网络服务通常是由许多应用程序接口（API）所组成的，它们通过网络，例如国际互联网（Internet）的远程服务器端，执行客户所提交服务的请求。

通常意义上 Web 服务是指在客户-服务器架构（Client-server）中根据 SOAP 协议来传递 XML 格式的消息。无论是定义还是实现 Web 服务，Web 服务由服务器提供一个机器可读的描述（通常基于 WSDL）来辨识服务器所提供的 Web 服务。此外，虽然 WSDL 不是 SOAP 服务端的必要条件，但在目前基于 Java 的主流 Web 服务开发框架中往往需要借助 WSDL 来实现客户端的源代码生成，因而一些工业标准化组织（WS-I，The Web Services Interoperability Organization，Web 服务互操作性组织）在 Web 服务定义中强制包含了 SOAP 和 WSDL。这里简单地解释一下相关的术语。

SOAP，即 Simple Object Access Protocol（简单对象访问协议），它是一种标准化的通信规范，主要用于 Web 服务中。SOAP 的出现是为了简化网页服务器（Web Server）在从 XML 数据库中提取数据时，无须花时间去格式化页面，并能够让不同应用程序之间通过 HTTP 通信协定，以 XML 格式互相交换彼此的数据，使其与编程语言、平台和硬件无关。用一个简单的例子来说明 SOAP 使用过程：一个 SOAP 消息可以发送到一个具有 Web 服务功能的 Web 站点，例如一个含有房价信息的数据库，消息的参数中标明这是一个查询消息，那么此站点就将返回一个 XML 格式的信息，其中包含了查询结果（价格、位置、特点或者其他信息）。由于数据是用一种标准化的可分析的结构来传递的，所以可以直接被第三方站点所利用。

WSDL，即 Web Services Description Language（Web 服务描述语言），这是一个基于 XML 的关于如何与 Web 服务通信和使用的服务描述；也就是描述与 Web 服务进行交互时需要绑定的协议和信息格式。通常采用抽象语言描述该服务支持的操作和信息，使用的时候再将实际的网络协议和信息格式绑定给该服务。WSDL 通常用来辅助生成服务器和客户端代码及配置信息。

UDDI，即 Universal Description，Discovery，and Integration（统一描述、发现和集成），它是一个基于 XML 的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务，应用程序可借由此协议在设计或运行时找到目标 Web 服务。

另外，为了提高 Web 服务间的互操作能力，WS-I 还特别发布了 Web 服务协议集（Profile）。协议集包含了一系列特定版本的核心定义（诸如 SOAP 和 WSDL），以及对其使用上的限制与约束。WS-I 还发布了用于部署协议集兼容 Web 服务的测试工具及相关用例。

为扩展 Web 服务能力，一些新的标准已经或正在被开发。这些标准通常被冠以 WS 字头（Web Service 的简称），以下是一个 WS 系列追加标准的不完全列表：

- WS 安全（WS-Security）。

定义了如何在 SOAP 中使用 XML 加密或 XML 签名来保护消息传递。可作为 HTTPS 保护的一种替代或扩充。

- WS 信赖性（WS-Reliability）。

一个来自 OASIS(结构化信息标准促进组织)的标准协议,用来提供可信赖的 WEB 服务间消息传递。

- WS 可信赖消息(WS-ReliableMessaging)。

同样是一个提供信赖消息的协议,由 Microsoft,BEA 和 IBM 发布。目前 OASIS 正对其实施标准化工作。

- WS 寻址(WS-Addressing)。

定义了在 SOAP 消息内描述发送/接收方地址的方式。

- WS 事务(WS-Transaction)。

定义事务处理方式。

简而言之,与 HTTP 通信方式相比,Web Service 最大的不同就在于它可以实现远程方法的调用而 HTTP 不能。

在 8.2.3 节中使用 HttpURLConnection 通信的例子实际上可以认为是使用了由人人网提供的一个简单的 Web 服务,只不过没有使用到 SOAP 协议,而是直接使用 POST 方法传送数据,这个服务即一个用户登录程序,在例子中我们将用户名和口令以参数的形式传递给远程的 Web 服务,远程 Web 服务处理了这个调用,然后将结果返回给客户端,从而使得客户端获取到了请求的页面。

8.4.2 Web 服务的使用方式

Web 服务实际上可以被看作是一组工具,它提供了多种方法供客户程序调用。使用 Web 服务有 3 种最普遍的方式,分别是:

- 远程过程调用(RPC)。
- 面向服务架构(SOA)。
- 表述性状态转移(Representational State Transfer,REST)。

下面分别对这 3 种方式进行简要介绍。

1. 远程过程调用

远程过程调用即是指 Web 服务提供一个分布式函数或方法接口供用户调用,这是一种比较传统的方式。通常,在 WSDL 中对 RPC 接口进行定义。

尽管最初的 Web 服务广泛采用 RPC 方式部署,但针对其过于紧密的耦合性的批评声也随之不断。这是因为 RPC 式的 Web 服务实质上是利用一个简单的映射,从而把用户请求直接转化成为一个特定语言编写的函数或方法。目前多数服务提供商认定此种方式在未来将难有作为,因此在他们的推动下,WS-I 基本协议集(WS-I Basic Profile)已不再支持远程过程调用。

2. 面向服务架构

现在,业界比较关注的是遵从面向服务架构(Service-oriented architecture,SOA)的概念来构建 Web 服务。在面向服务架构中,通信由消息驱动,而不再是某个动作(方法调用)。这种 Web 服务也被称作面向消息的服务。

SOA 式 Web 服务得到了大部分主要软件供应商以及业界专家的支持和肯定。作为与

RPC 方式的最大差别，SOA 方式更加关注如何去连接服务而不是去特定某个实现的细节。WSDL 定义了联络服务的必要内容。

3. 表述性状态转移

表述性状态转移（Representational State Transfer，REST）式的 Web 服务类似于 HTTP 或其他类似协议，它们把接口限定在一组广为人知的标准动作中（比如 HTTP 的 GET、PUT、DELETE）以供调用。此类 Web 服务关注与那些稳定的资源的互动，而不是消息或动作。

此种服务可以通过 WSDL 来描述 SOAP 消息内容，通过 HTTP 限定动作接口；或者完全在 SOAP 中对动作进行抽象。

8.4.3 Android 使用 Web 服务

目前由世界上各种组织、企业或者个人提供的 Web 服务越来越多，小到由一个初学者就可以发布的用于查询时间的 Web 服务，大到类似于 Amazon 所提供的用于支持云计算的 Web 服务，各种各样的 Web 服务提供了丰富而简便的数据获取方式。常见的一些 Web 服务有：

- 天气预报。
- IP 地址所在地查询。
- 手机号码归属地查询。
- 邮政编码与地址双向查询。
- 验证码图片。
- 中文简体与繁体的转换。
- 中英文双向翻译。
- 列车时刻表。
- 即时外汇汇率数据。
- 航班时刻表。

……

可以发现，只要是涉及信息发布或者数据计算等的操作都可以做成 Web 服务，读者可以在 www.webxml.com.cn 上找到更多的 Web 服务，每一种 Web 服务都提供了若干相应的操作。

本节将在 Android 平台上演示如何使用“输入城市名查询天气预报”这个 Web 服务，由 www.webxml.com.cn 提供的天气预报 Web 服务的说明页面在：http://www.webxml.com.cn/webservices/weatherwebservice.asmx。

读者可以通过这个页面对该 Web 服务所提供的一些操作及具体使用方式进行全面的了解，这里简要地对这个 Web 服务所提供的一些操作进行说明。

- getSupportCity。

该方法用于查询该天气预报 Web 服务所支持的国内外城市或地区的信息。

输入参数：byProvinceName ＝ 指定的洲或国内的省份，若为 ALL 或空，则表示返回全部城市。

返回数据：一个一维字符串数组 String()，结构为：城市名称(城市代码)。例如如果输入参数 byProvinceName ＝ 四川，则会返回(只给出部分)：

```
<string>成都 (56294)</string>
<string>泸州 (57602)</string>
<string>内江 (57504)</string>
<string>凉山 (56571)</string>
```

- getSupportDataSet。

该方法用于获取该天气预报 Web Services 支持的洲、国内外省份和城市信息，不需要输入参数，直接返回完整的支持列表及相关数据库属性。

- getSupportProvince。

该方法用于获取该天气预报 Web Services 支持的洲、国内外省份和城市信息，类似于前一种方法，不过该方法仅返回所支持的省名或者地区名。

- getWeatherbyCityName。

该方法用于根据城市或地区名称查询获得未来 3 天内的天气情况、现在的天气实况、天气和生活指数等信息。

在如上的这几个方法中，最后一个方法的作用就是用于根据城市名称查询天气预报，在即将完成的示例中，主要就是通过这个方法来获取数据并对其进行解析，从而得到天气预报的信息。在使用这个 Web 服务之前，首先完成在 8.2.2 节中留下的问题，即 Google Weather 服务所返回的 xml 解析工作，这相当于使用 Http Get 的方式来访问 Web 服务；然后，再在稍后介绍如何使用 SOAP 的方式来访问 Web 服务。

1. 通过 Http Get 方式

1) 示例运行效果

首先来看一下本示例的运行效果，如图 8-18 和图 8-19 所示，该配套示例为 WeatherApp。



图 8-18　初始状态



图 8-19　查询结果

图 8-18 是示例启动后的界面，在左上方的文本框中输入想要查询的城市名，例如成都，然后单击 Search 按钮，得到如图 8-19 所示的结果，结果中包括了当前的天气状况以及未来几天的天气预报信息。图 8-20 给出了本示例中设计的类和方法的结构和关系图。

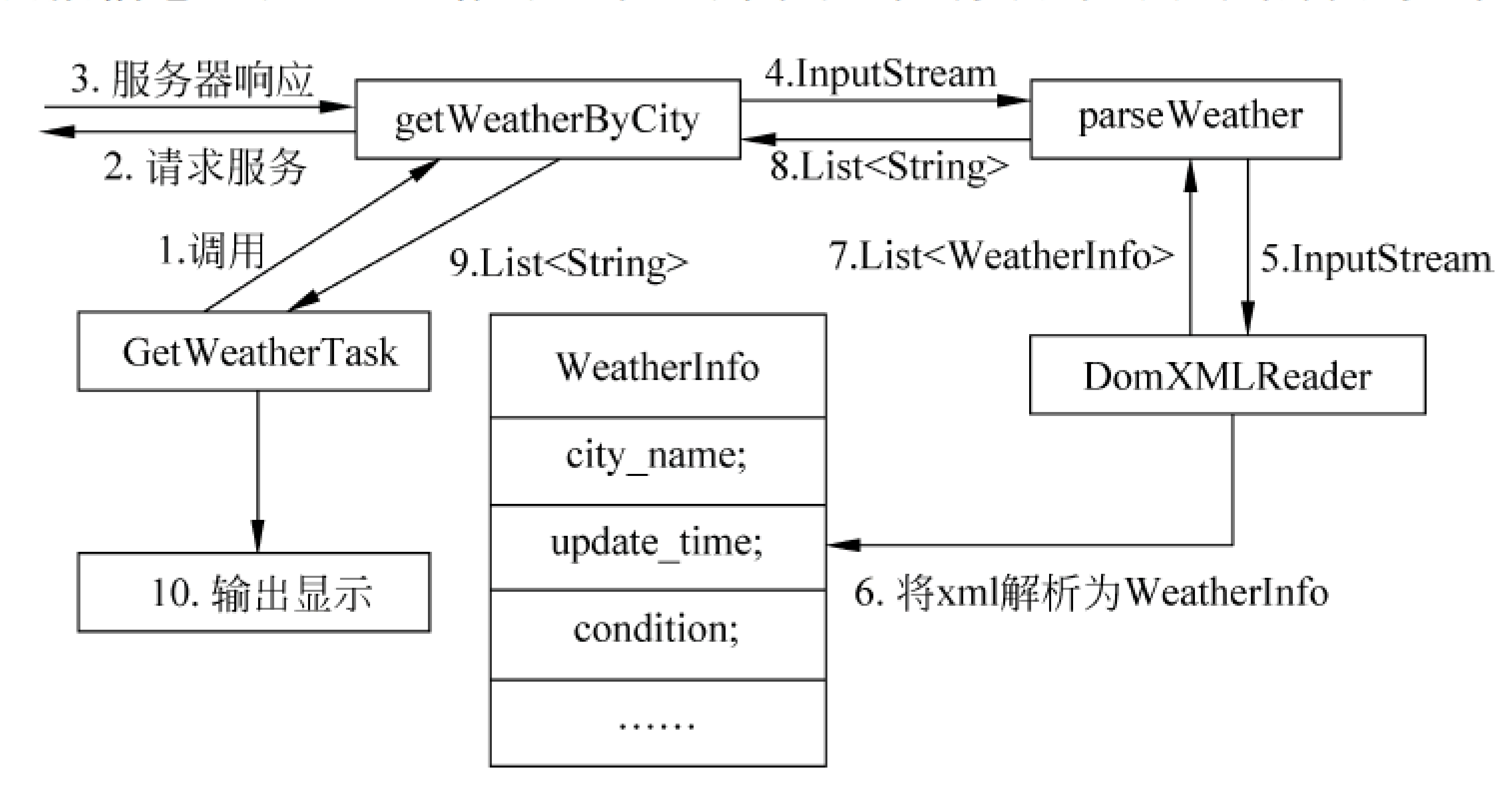


图 8-20　示例结构示意图

2）使用 Http Get 方式获取数据

在 8.2.2 节已经介绍了如何使用 Http Get 方式获取由服务器传回的数据，这里使用的方法与前面介绍的基本一样，唯一不同的就是不再简单地将服务器返回的数据当作字符串来处理，而是获取为一个输入流(InputStream)供另一个专有的方法进行解析。获取天气数据的代码段如下：

```
/**
 * 从 Google Weather API 获取数据
 * @param city 城市名称或邮政编码
 * @return 天气数据信息
 */
public List<String> getWeatherByCity(String city) {
    HttpClient httpClient = new DefaultHttpClient();
    HttpContext localContext = new BasicHttpContext();
    HttpGet httpGet = new HttpGet(GOOGLE_API_URL + city);
    Log.v(TAG, GOOGLE_API_URL + city);
    try {HttpResponse response = httpClient.execute(httpGet, localContext);
        if (response.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
            Log.e(TAG, "Failed. status code is not ok.");
            httpGet.abort();
        }
        else {
            HttpEntity httpEntity = response.getEntity();
            String string = EntityUtils.toString(httpEntity, "utf-8").trim();
            InputStream is = new ByteArrayInputStream(string.getBytes("utf-8"));
            return parseWeather(is);
        }
    } catch (Exception e) {
        Log.e(TAG, "Failed to get weather", e);
    } finally {
        httpClient.getConnectionManager().shutdown();
```

```
26         }
27         List<String> error = new ArrayList<String>();
28         error.add(NETWORK_ERROR);
29         return error;
30     }
```

其中第 8 行使用到了新类 HttpContext，这个类的作用是将 Http 操作的请求、响应及其他的一些相关数据捆绑在一起，以供其他的方法使用；第 17～20 行则是根据服务器的响应获取其中的数据输入流，然后将这个输入流传递给 parseWeather 方法解析，在 parseWeather 方法(接下来将会介绍)解析完成时将返回一个 List<String>类型的字符串列表，这个字符串列表包含了一系列的天气预报数据，将在随后被进一步处理之后显示在 Activity 界面上。

3）解析服务器返回的数据

通过 Http Get 方法获取到服务器的响应之后，就需要对这个输入流进行解析，为此，实现了一个用于存放天气预报信息的类 WeatherInfo，这个类封装了一系列的成员变量和它们对应的 get 和 set 方法，另外还实现了一个文档解析类 DomXMLReader，输入流最终传入到这个类的静态方法 readXML 中进行解析，解析的结果是得到一个保存了所有天气预报信息的 List<WeatherInfo>列表，通过这个列表就能够在界面中显示出天气预报信息。

WeatherInfo 类包括了如下一些成员变量(get、set 方法略)，这些成员变量用于存储所有从输入流解析出来的相关信息：

```
public class WeatherInfo {
    String city_name;           //城市名
    String update_time;         //天气更新时间

    String condition;           //天气说明
    String temp_f;              //华氏温度
    String temp_c;              //摄氏温度
    String humidity;            //湿度
    String icon;                //天气图标
    String wind_condition;      //风向风速信息

    String day_of_week;         //星期几
    String temp_low_f;          //最低温度(华氏(英文 api)或者摄氏(中文 api))
    String temp_high_f;         //最高温度(华氏(英文 api)或者摄氏(中文 api))
}
```

用于将服务器响应解析为 WeatherInfo 列表的类 DomXMLReader 代码如下(为了节约篇幅，这里仅列出解析出当前天气信息的代码，用于解析当前预报城市的相关信息及解析预报信息的代码与之类似)：

```
public static List<WeatherInfo> readXML(InputStream inStream) {
    List<WeatherInfo> weatherReportList = new ArrayList<WeatherInfo>();
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
```

```
06            Document dom = builder.parse(inStream);
07            Element root = dom.getDocumentElement();
08            //解析出当前天气信息
09            NodeList items = root.getElementsByTagName("current_conditions");
10            for (int i = 0; i < items.getLength(); i++) {
11                WeatherInfo currentCondition = new WeatherInfo();
12                Element currentNode = (Element) items.item(i);
13                NodeList childsNodes = currentNode.getChildNodes();
14                for (int j = 0; j < childsNodes.getLength(); j++) {
15                    Node node = (Node) childsNodes.item(j);
16                    if(node.getNodeType() == Node.ELEMENT_NODE){
17                        Element childNode = (Element) node;
18                        if ("condition".equals(childNode.getNodeName())) {
19                            //获取 condition 节点的 data 属性值,即天气信息
20                            currentCondition.setCondition(childNode
21                              .getAttribute("data"));
22                        } else if ("temp_f".equals(childNode.getNodeName())) {
23                            //获取 temp_f 节点的 data 属性值,即华氏度
24                          currentCondition.setTemp_f(childNode.getAttribute("data"));
25                        } else if ("temp_c".equals(childNode.getNodeName())) {
26                            //获取 temp_c 节点的 data 属性值,即摄氏度
27                            currentCondition.setTemp_c(childNode.getAttribute("data"));
28                        } else if ("humidity".equals(childNode.getNodeName())) {
29                            //获取 humidity 节点的 data 属性值,即湿度
30                            currentCondition.setHumidity(childNode.getAttribute("data"));
31                        } else if ("icon".equals(childNode.getNodeName())) {
32                            //获取 iconPath 节点的 data 属性值,即图片 url
33                            currentCondition.setIconPath(childNode.getAttribute("data"));
34                        } else if ("wind_condition".equals(childNode.getNodeName())) {
35                            //获取 wind_condition 节点的 data 属性值,即风向风速信息
36                            currentCondition.setWind_condition(childNode
37                              .getAttribute("data"));
38                        }
39                    }
40                }
41                weatherReportList.add(currentCondition);
42            }
43          inStream.close();
44        } catch (Exception e) {
45            e.printStackTrace();
46        }
47        return weatherReportList;
48  }
```

该解析类实际上就是一个标准的 xml 文档解析器,根据 xml 的标签和属性来获取指定的数据,代码第 09 行通过 getElementsByTagName("current_conditions")方法获取了 xml 文档中所有名为 current_conditions 的标签列表,然后在第 10～42 行中的 for 循环体中对这个列表中的所有标签进行解析,依次获取了天气信息、华氏温度、摄氏温度、湿度、天气图标和风向风速等信息(代码第 18～35 行),并且将得到的 WeatherInfo 对象添加到列表 weatherReportList 中(代码第 41 行),当文档解析完毕之后,返回得到的天气预报列表(weatherReportList)(代码第 47 行)如图 8-20 所示,在 getWeatherByCity 方法中拿到了服

务器响应的输入流之后，调用了 parseWeather 方法来对这个输入流进行处理，parseWeather 方法代码如下：

```
/**
 * 根据解析结果得出天气数据
 * @param inputstream 由 Google Weather Web 服务返回的输入流
 * @return 天气数据
 * @throws IOException
 */
public List<String> parseWeather(InputStream inputstream) throws IOException {
    List<WeatherInfo> list = DomXMLReader.readXML(inputstream);
    List<String> weatherInfos = new ArrayList<String>();
    WeatherInfo weather_report = list.get(0);
    if(weather_report.getIconPath().endsWith(".gif"));{
        try
        {
            bitmapList.add(getNetBitmap("http://www.google.com"
                + weather_report.getIconPath()));
        }
        catch (Exception e)
        {Log.e(TAG, "IOException");}
    }
    times.add("现在: ");
    String without_city_name_and_time = "天气: "
       + new String(weather_report.getCondition()
            .getBytes(), "utf-8") + ", " + "华氏: " + weather_report.getTemp_f() + "℉\n"
         + weather_report.getHumidity() + ", " + weather_report.getWind_condition();
    weather_report = list.get(1);
    String city_name_and_time = "城市: " + new String(weather_report.getCity_name()
        .getBytes(), "utf-8") + "\n";
    weatherInfos.add(city_name_and_time + without_city_name_and_time);
    for(int i = 2; i <= 5; i++){
           weather_report = list.get(i);
           times.add(weather_report.getDay_of_week());
           if(weather_report.getIconPath().endsWith(".gif"));{
               try{
                   bitmapList.add(getNetBitmap(http://www.google.com
                     + weather_report.getIconPath()));
               }
               catch (Exception e){Log.e(TAG, "IOException");
             }
        }
        String forecastCondition = "天气: "
             + new String(weather_report.getCondition().getBytes(), "utf-8") + "\n"
             + "最低: " + weather_report.getTemp_low_f() + "℃\n"
             + "最高: " + weather_report.getTemp_high_f() + "℃\n";
        weatherInfos.add(forecastCondition);
    }
    return weatherInfos;
}
```

parseWeather 方法首先就使用了 DomXMLReader 来解析 xml 得到一个 WeatherInfo 的列表(代码第 08 行),接下来的代码的作用就是将这些 WeatherInfo 对象所包含的信息“转译”为一个 String 类型的列表,实际上就是一个组装信息并使之变得可读的过程,由于 Google 的天气服务的天气图标可以直接从网络上实时下载,因此还使用到了一个 Bitmap 类型的列表(代码第 11～19 行),这个列表依次存放着天气图标,并且与保存天气预报信息的列表顺序一致,其中第 10～27 行代码将当前的天气信息及城市信息组装成了用户易读的信息,相似地,第 28～45 行代码则用于将未来几天的天气预报信息组装成用户易读的信息。

4) 显示天气信息

经过上面一系列的方法处理之后,由服务器返回的 xml 文档已经被转换为用户易读的信息了,剩下的工作就是将这些可读的信息输出显示到界面上,由于在显示的时候还需要通过网络下载图标资源,为了防止用户界面被阻塞,采用了 AsyncTask 这个可异步执行的类,在前面 4.2.4 节已经对其进行了说明,读者可以返回去阅读一下该节的内容。此处通过继承 AsyncTask 实现了一个用户异步更新用户界面显示的 GetWeatherTask 类,其代码如下:

```
//采用异步任务更新显示
class GetWeatherTask extends AsyncTask<String, Integer, List<String>> {

    @Override
    protected List<String> doInBackground(String... params) {
        String city = params[0];
        //调用 Google 天气服务查询指定城市的当日天气情况
        return getWeatherByCity(city);
    }
    protected void onPostExecute(List<String> result) {
        if(result.size() != 5 || times.size() != 5
                || bitmapList.size() != 5 || times == null) {
            //数据异常
            return;
        }
        //把 doInBackground 处理的结果即天气信息显示到界面
        tvWeatherInfo.setText(result.get(0));
        time.setText(times.get(0));
        icon.setImageBitmap(bitmapList.get(0));
        bitmapList.clear();
        times.clear();
    }
}
```

如上面的代码所示,该任务的工作流程是:当用户开启一个 GetWeatherTask 任务后(即单击查询按钮),首先会在 doInBackground 方法中执行用于获取天气预报相关信息的方法 getWeatherByCity,当 doInBackground 执行完成之后,onPostExecute 方法会自动被调用,并且将解析完成的 List＜String＞作为参数传入,最后在 onPostExecute 方法中更新界面显示,从而实现天气预报信息的显示。

2. 通过 SOAP 方式

本节将实现通过 SOAP 方式来访问由 www.webxml.com.cn 提供的天气预报 Web 服

务，在前面已经对该天气服务所提供的一些接口进行了介绍，本节将利用其提供的 getWeatherByCityName 接口并且通过 SOAP 的方式来使用该 Web 服务。到如下页面可以了解到 getWeatherByCityName 接口的使用方法：

```
http://www.webxml.com.cn/webservices/weatherwebservice.asmx?op=getWeatherbyCityName
```

从上面所示的页面中可以找到有关 getWeatherByCityName 接口的详细使用说明，包括输入参数类型以及返回数据的组织方式（返回的字符串数组中每一位的含义），另外还提供了天气图标资源的下载，将天气图标下载到本地之后，就不必再像前面采用 Http Get 方式时到网络上实时下载天气图标了。这样做的优点是加快了信息的加载速度，缺点是占用本地资源并且天气图标的更新不方便，因此，读者也可以根据自己的爱好去动手制作自己喜爱的图标，只要在解析数据的时候将图标文件与用于指定图标的字符串相对应即可。

要在 Android 上使用 SOAP 协议，需要使用一个由 KSOAP 2 提供的 Java 类库，KSOAP 2 是一个适用于受限 Java 环境的 SOAP Web 服务客户端类库，使用这个类库可以很方便地通过 SOAP 方式使用 Web 服务。KSOAP 2 的主页地址为 http://ksoap2.sourceforge.net/，读者可以到这个页面上了解更多的信息。KSOAP 2 的 Android 版本项目主页可以在 Google Code 上找到，地址为 http://code.google.com/p/ksoap2-android/。

在 KSOAP 2 的 Android 版本项目的 Wiki 页面：http://code.google.com/p/ksoap2-android/wiki/HowToUse 可以找到适用于 Eclipse/ADT 开发的 Java 类库，本示例中使用的是最新发布的版本，jar 包名为：

```
ksoap2-android-assembly-2.6.0-jar-with-dependencies.jar
```

下载的地点如图 8-21 所示。



图 8-21 KSOAP2-Android 下载地点

KSOAP2 的在线 API 文档地址在 http://ksoap2.sourceforge.net/doc/api/，在接下来给出的示例中，仅对使用到的一些类和方法作简要说明，详细的信息请读者到上述网址查阅。

1）示例运行效果

配套示例为 WeatherAppSOAP。示例的运行效果如图 8-22 和图 8-23 所示，为了使代码结构清晰，仅仅实现了对当天天气信息的查询，读者如果想要实现类似于 8.4.3 节第 1 部分的多天天气预报的效果，可以参照 getWeatherByCityName 方法的返回数据说明，在示例的基础上进行简单的扩展就可以了。图 8-24 给出了本示例中设计的类和方法的结构和关系图。



图 8-22 查询成都的天气



图 8-23 查询上海的天气

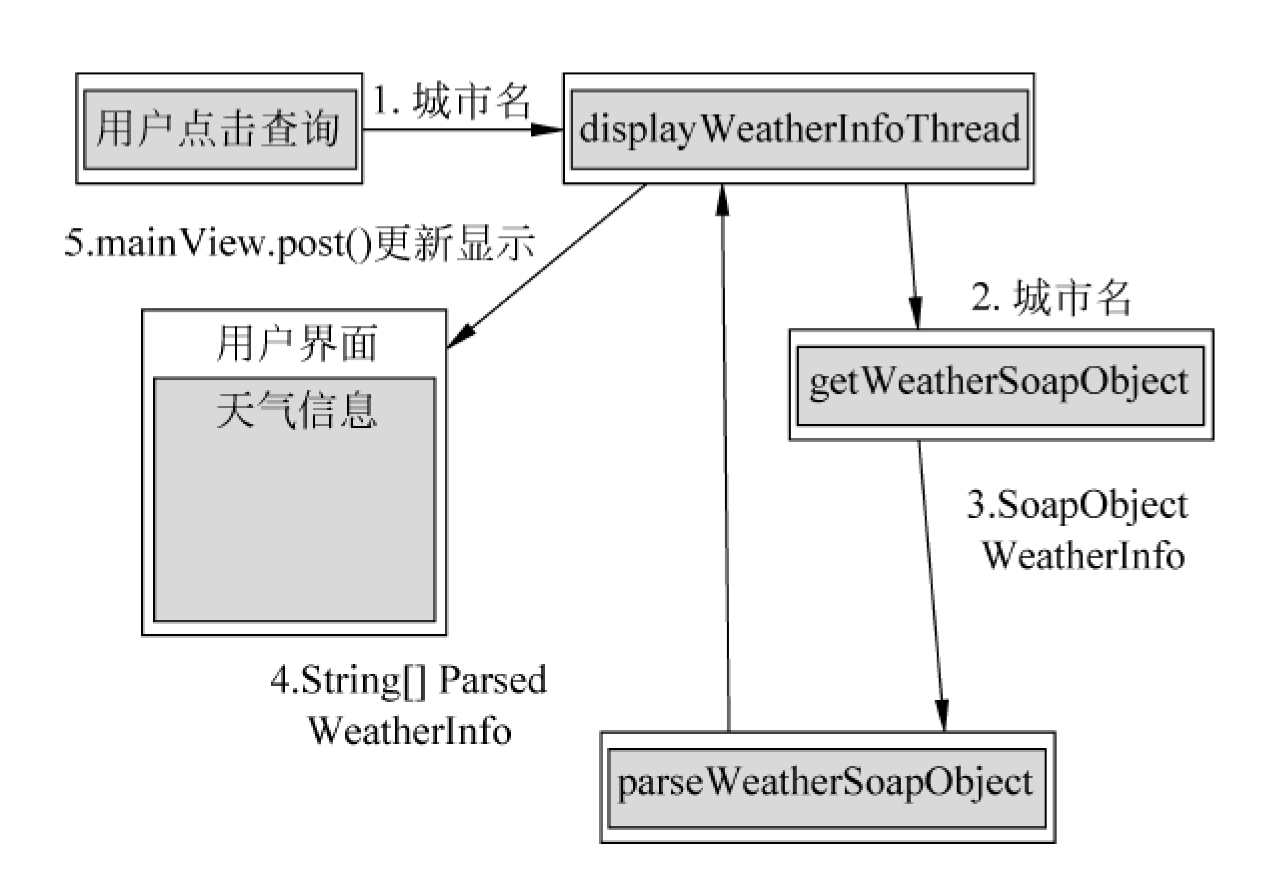


图 8-24 示例结构示意图

2）使用 SOAP 方式获取数据

使用 SOAP 方式从 Web 服务处获取数据使用到了由 KSOAP 2 提供的如下几个类：

```
import org.ksoap2.SoapEnvelope;
import org.ksoap2.serialization.SoapObject;
import org.ksoap2.serialization.SoapSerializationEnvelope;
import org.ksoap2.transport.HttpTransportSE;
```

第一个类 SoapEnvelope 封装了一次 SOAP 请求所需要的相关数据，并且也负责接收由服务器发回的响应，它们分别对应了 SoapEnvelope.bodyIn 和 SoapEnvelope.bodyOut 这两个成员变量；第二个类 SoapObject 用于构建一次 SOAP 请求以及封装由服务器返回的数据，在本例中 SoapEnvelope.bodyIn 和 SoapEnvelope.bodyOut 都是使用的 SoapObject 对象；第三个类 SoapSerializationEnvelope 继承自 SoapEnvelope，加入了用于序列化 Soap 的相关功能；HttpTransportSE 则提供了方便在 HTTP 上使用 J2SE 通用连接框架来进行

SOAP 调用的方法。使用 SOAP 方式请求 Web 服务的代码如下：

```
01 //从 Web 服务处请求数据
02 public SoapObject getWeatherSoapObject(String cityName) {
03     try {
04         SoapObject requestmessage = new SoapObject(NAMESPACE,METHOD_NAME);
05         requestmessage.addProperty("theCityName", cityName);
06 
07         //创建用于封装 SOAP 请求和响应的对象,并设置请求消息
08         SoapSerializationEnvelope envelope =
09             new SoapSerializationEnvelope(SoapEnvelope.VER11);
10         envelope.bodyOut = requestmessage;
11         //兼容.Net Web 服务的默认编码,该 Web 服务是基于.Net 的
12         envelope.dotNet = true;
13         //向 Web 服务发送请求
14         HttpTransportSE sendRequest = new HttpTransportSE(WS_URL);
15         sendRequest.call(SOAP_ACTION, envelope);
16 
17         //获取 Web 服务的响应
18         SoapObject result = (SoapObject) envelope.bodyIn;
19         SoapObject weatherInfo = (SoapObject)
20             result.getProperty("getWeatherbyCityNameResult");
21         return weatherInfo;
22     } catch (Exception e) {
23         e.printStackTrace();
24         return null;
25     }
26 }
```

在上面的代码中，第 04 行和第 05 行构建了一个用于封装对 Web 服务的请求的 SoapObject；第 07～11 行创建了用于封装 SOAP 请求和响应的对象，并将前面构建的 SoapObject 设置为请求消息；第 13～15 行创建了一个 HttpTransportSE 对象并通过它向 Web 服务发出请求；第 17～20 行则是获取 Web 服务的响应并将其封装为一个 SoapObject 对象并作为方法的返回值返回。取得了这个封装有天气信息的 SoapObject 对象之后，通过对它的解析就能够得到可读性强的天气预报信息。

3）解析服务器返回的数据

解析工作由 parseWeatherSoapObject 方法来完成，该方法对传入的 SoapObject 对象进行解析，组装成可读性强的天气信息，其代码如下：

```
//解析 Web 服务响应
private String[] parseWeatherSoapObject(SoapObject weatherInfo) {
    //组装天气信息
    String weatherToday =  "发布时间：" + weatherInfo.getProperty(4) + "\n"
            + "今日天气：" + weatherInfo.getProperty(6) + "\n"
            + "气温范围：" + weatherInfo.getProperty(5) + "\n"
            + "风向风速：" + weatherInfo.getProperty(7) + "\n";
    //获取代表天气趋势开始时状况的图标的字符串(多云,阴,雨等)
    String weatherNow = weatherInfo.getProperty(8).toString();
    //获取代表天气趋势结束时状况的图标的字符串
    String weatherThen = weatherInfo.getProperty(9).toString();
```

```
12      //准备返回值并返回
13      String[] parsedWeatherInfo = new String[3];
14      parsedWeatherInfo[0] = weatherToday;
15      parsedWeatherInfo[1] = weatherNow;
16      parsedWeatherInfo[2] = weatherThen;
17      return parsedWeatherInfo;
18  }
```

该方法的第03～07行代码取出了SoapObject中与今天的天气信息相关的一些字段的值，然后为它们加上能够说明其含义的解释文字即得到了可读性强的天气信息，每个字段的具体含义可以在前面提到的网页上查询到；第08～11行代码则是取出了另外两个代表天气图标的字符串，它们分别是今日天气趋势的开始和结束时的天气状况（例如晴转多云、大雨转阴这样的趋势），这两个字段的值通常是一个类似于“1.gif”的字符串，它们并不是直接输出显示的，而是通过另一个名为setIcon的方法来根据字段的值在指定的位置显示相应的天气图标，setIcon方法的一部分代码如下：

```
//根据代表天气状况的字符串设置天气图标
private void setIcon(String weather, ImageView weatherIconView) {
    if(weather == null) return;
    if(weather.equalsIgnoreCase("nothing.gif"))
        weatherIconView.setBackgroundResource(R.drawable.a_nothing);
    if(weather.equalsIgnoreCase("0.gif"))
        weatherIconView.setBackgroundResource(R.drawable.a_0);
//…………
}
```

parseWeatherSoapObject方法的第12～17行代码将前面获取的数据存放在一个字符串数组中并且作为返回值返回，这个字符串数组用于显示的代码使用。

4）显示天气信息

为了不阻塞用户界面，显示天气信息的功能同样由一个线程类来完成，在这个线程类中依次调用了getWeatherSoapObject和parseWeatherSoapObject方法，从而得到包含有天气信息数据的字符串数组，然后分别对界面上的TextView和ImageView赋值，线程类DisplayWeatherInfoThread的代码如下：

```
//显示当前的天气信息
class DisplayWeatherInfoThread extends Thread{
    @Override
    public void run(){
        String cityName = inputCityName.getText().toString();
        SoapObject weatherInfo = getWeatherSoapObject(cityName);
        final String[] parsedWeatherInfo = parseWeatherSoapObject(weatherInfo);

        mainView.post(new Runnable(){
            @Override
            public void run() {
                String weatherToday = parsedWeatherInfo[0];
                resultTextView.setText(weatherToday);
```

```
14
15                  String weatherNow = parsedWeatherInfo[1];
16                  setIcon(weatherNow, weatherNowIcon);
17
18                  String weatherThen = parsedWeatherInfo[2];
19                  setIcon(weatherThen, weatherThenIcon);
20              }
21          });
22      }
23  }
```

用户每单击一次“查询当前天气”按钮，就将开启一个新的 DisplayWeatherInfoThread 线程，然后由该线程负责查询 Web 服务并且显示结果到用户界面上。代码第 05 行获取了用户输入，第 06 行和第 07 行调用了发送 SOAP 请求的两个方法并得到结果字符串数组 parsedWeatherInfo；第 09～21 行则是从字符串数组中取出数据并显示到 TextView 和 ImageView 上，显示 ImageView 使用了 setIcon 方法，由于 Android 的单线程模型(见第 4 章)，在非 UI 线程中不能够直接对视图进行操作，因此使用了 View.post()方法将更改视图的操作传递给 UI 线程执行。

8.5 WebView

8.5.1 WebView 简介

WebView，顾名思义它就是一个用于显示 Web 页面的视图(View)，可以把它看成是一个类似于 TextView 或者 ImageView 的控件，不过它却拥有比一般的控件更加强大的功能，它不像 TextView 之类的控件来自于 android.widget 包，而是来自于 android.webkit 包，因此，它可以被看做是一个微型的浏览器，在默认状态下，WebView 不具备通常浏览器常有的一些功能组件例如前进、后退、刷新等功能，仅为开发者提供了一个视图，仅此而已，但正是因为如此，使得我们在使用 WebView 时拥有了极高的自由度。WebView 虽然不是浏览器，但是它却提供了浏览器所需要的大部分核心功能，它使用 WebKit 作为 Web 页面的渲染引擎，在它的基础之上甚至可以实现一个属于自己的浏览器，或者也可以仅仅使用它在 Activity 的某处显示一个指定的网页内容，也可以为它实现浏览器通常所具备的前进、后退、缩放视图或者文本搜索的功能。

在默认情况下，WebView 没有开放 JavaScript 支持的功能，并且会忽略掉所有的 Web 页面上的错误而不做任何提示，WebView 最基本的也是最常用的用途就是作为应用程序界面的一部分，在这个部分里显示 Html 页面。在这种模式下，用户仅仅需要查看 WebView 中的内容而不需要与之发生交互，WebView 中的内容也不需要获取用户的交互，Android 开发文档建议我们仅在有这样的需求时才使用 WebView，如果需要的功能是类似于一个成熟而完整的浏览器所提供的功能，那么请使用 Intent 的方式来调用系统默认的 Web 浏览器来进行处理，类似于如下的方法：

```
Uri uri = Uri.parse("http://www.example.com");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

因此，通常仅在如下两种情况下使用 WebView（除非本身就是为了实现一个完整的浏览器）：

- 作为 Activity 界面的一部分显示给用户，不进行任何交互。
- 实现 Web App（Web 应用）。

使用 WebView，可以通过 url 打开远程 Web 页面，也可以加载存放于本地的 Html 文档，当 WebView 的 JavaScript 支持功能开启后，还能够使用 JavaScript 来处理与用户的交互，进而可以实现 Java 代码与 JavaScript 代码之间的交互。WebView 类提供了许多方法，常用的有以下一些：

- canGoBack()——如果当前页面存在可以返回的历史项，则该方法的返回值为 true。
- canGoForward()——如果当前页面存在可以进行到的历史项，则该方法的返回值为 true。
- canZoomIn()——当前的 Web 页面可以放大。
- canZoomOut()——当前的 Web 页面可以缩小。
- capturePicture()——将当前的 WebView 截图并且返回一个 Picture 对象。
- getSettings()——该方法将会返回一个 WebObject 对象，这个对象用于控制当前 WebView 的一些设置属性。
- getTitle()——获取当前页面的标题。
- getUrl()——获取当前页面的 url。
- goBack()——返回上一页。
- goForward()——前进到下一页。
- loadData()——加载指定的数据到 WebView。

……

更多的方法请读者到 Android 的开发文档中进行查看，另外一个重要的类就是通过 getSettings()取得的 WebSettings，通过 WebSettings 对象可以对网页的字体类型、字体大小等属性进行设置，也可以通过它获取到 WebView 相关属性的当前值，从而完成相应的事件处理等。

8.5.2 使用 WebView 显示远程网页

为了让读者尽快熟悉 WebView，首先将带领读者一起来实现“使用 WebView 显示远程网页”的功能，通过这个示例让读者了解 WebView 最简单的使用方式（示例取自 Android 官方文档，配套的 Eclipse 项目为 HelloWebView）。示例的效果就是加载一个指定的 url 所链接到的网页内容，如图 8-25 所示。

图 8-25 就是使用 WebView 在 Activity 中显示的 Google 搜索首页（www.google.com.hk），为了实现如图 8-25 所示的效果，需要经过如下一系列的步骤：

（1）在 Eclipse 中创建一个名为 HelloWebView 的项目。



图 8-25　使用 WebView 加载网页的效果

（2）首先需要修改的是项目下的 res/layout/main.cml 文件，在 Eclipse 中打开该文件，然后修改代码如下：

```
01  <?xml version="1.0" encoding="utf-8"?>
02  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03      android:layout_width="fill_parent"
04      android:layout_height="fill_parent"
05      android:orientation="vertical" >
06
07      <WebView
08          xmlns:android="http://schemas.android.com/apk/res/android"
09          android:id="@+id/webview"
10          android:layout_width="fill_parent"
11          android:layout_height="fill_parent" />
12
13  </LinearLayout>
```

修改的实际上是第 07～11 行代码，将原有的一个 TextView 替换成了 WebView。

（3）然后打开 HelloWebViewActivity.java 文件，在 HelloWebViewActivity 类的开始处添加 WebView 对象的声明：

```
WebView mWebView;
```

然后修改 onCreate()方法代码如下：

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
03        setContentView(R.layout.main);
04
05        mWebView = (WebView) findViewById(R.id.webview);
06        mWebView.getSettings().setJavaScriptEnabled(true);
07        mWebView.loadUrl("http://www.google.com.hk");
08    }
```

这段代码的作用是使用 main.xml 布局文件中声明的 WebView 来初始化 mWebView 成员变量(代码第 05 行),然后通过 mWebView 的 getSetting()方法获取到该 WebView 的设置对象 WebSettings,并且使用 setJavaScriptEnabled(boolean)方法使得该 WebView 支持 JavaScript,最后,使用 loadUrl(String)方法加载一个 Web 页面并显示到 mWebView 上。

(4) 由于新建的项目默认没有声明使用网络的权限,而该应用需要使用网络来加载页面,因此需要在项目的 AndroidManifest.xml 文件中添加声明:

```
<uses-permission android:name="android.permission.INTERNET" />
```

(5) 细心的读者可以发现如图 8-25 所示的应用界面是不存在标题栏的,这是为了使得网页有更大面积的屏幕可以显示,为了实现这个效果,需要在 AndroidManifest.xml 文件中将 HelloWebViewActivity 的主题声明为 NoTitleBar:

```
<activity android:label="@string/app_name" android:name=".HelloWebViewActivity"
    android:theme="@android:style/Theme.NoTitleBar" >
```

现在运行该应用,就能够得到如图 8-25 所示的效果,然而这个应用的作用也仅仅限于显示这个指定的网页,如果点击该网页上的链接,则会触发使 Android 自带的 Web 浏览器开启的 Intent,然后在打开的浏览器中显示新的链接,这是因为目前所实现的这个 Activity 还不具备响应这个 URL 跳转事件的功能。那么如何才能够使得当点击网页上的链接后,新链接到的页面仍在当前的 Activity 中打开呢?答案将在下一节给出。

8.5.3 为 WebView 添加功能

在 8.5.2 节中留下了一个问题,就是如何使得当点击 WebView 中的链接时,可以不去调用系统默认的浏览器处理,而是直接在当前的 WebView 中显示。当然,可以通过为应用添加一个 intent filter 来解决这个问题,但是这样会使得该应用影响到系统中所有打开网页的请求,这显然不是我们想要的效果,毕竟我们的目的不是去实现一个浏览器。这里采用另外一种方法——通过覆盖 WebView 类中的 WebViewClient 对象的方法来实现这个目标。WebView 默认的 WebViewClient 不会处理加载新链接的请求,为此,在 HelloWebViewActivity 类中添加一个内部类,并且重写相应的方法,然后使用这个内部类替换掉默认的 WebViewClient,该内部类的代码如下:

```
private class HelloWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
```

```
        view.loadUrl(url);
        return true;
    }
}
```

然后在 HelloWebViewActivity 的 onCreate()方法中通过 setWebViewClient()方法将 WebViewClient 设置为 HelloWebViewClient：

```
mWebView.setWebViewClient(new HelloWebViewClient());
```

这个方法必须在 mWebView 初始化之后调用，否则会抛出 NullPointer 异常。经过如上的设置之后，现在再点击链接时，新的 HelloWebViewClient 就会使得新的页面在同一个 mWebView 中加载，这是由于 shouldOverrideUrlLoading(WebView, String)方法将会捕获到点击链接这个事件，事件发生时将会传入当前的 WebView 对象和请求的 url 字符串，因此通过 view.loadUrl(url)方法就可以在当前页面中打开新的链接，该方法的返回值为 true 时代表当前的方法已经完成了对该事件的处理，因此不会再将这个事件转发出去；如果返回 false，则仍然会产生一个 Intent，从而触发系统默认的浏览器来处理这个事件。读者可以自行测试一下，如果此处将返回值设置为 false，虽然新的链接仍然会在当前的 WebView 中加载，而默认浏览器也会被打开并加载这个同样的链接页面。

这里简要说明一下在 WebView 下对 Back 按钮的响应，Back 按钮通常是 Android 手机上提供的少数实体按钮之一，单击这个按钮所发生的默认事件是返回到前一个 Activity 界面(具体的返回策略由 Android 系统管理)，为什么此处要提到这个按钮呢？这是因为如果读者现在再去运行刚刚修改过的 HelloWebView 应用，会发现点击链接可以继续在当前 WebView 中显示了，然而却没有办法返回到前一个页面，因为此时的 Back 按钮事件会导致 Activity 直接退出，为了完善 HelloWebView，使之具备返回上一个页面的功能，需要在 Activity 中添加对 Back 按钮事件的处理，并覆盖掉默认的 Back 按钮导致 Activity 退出的作用。为此，需要重写 HelloWebViewActivity 的 onKeyDown()方法，在 Activity 中添加如下代码：

```
01  @Override
02  public boolean onKeyDown(int keyCode, KeyEvent event) {
03      if ((keyCode == KeyEvent.KEYCODE_BACK) & mWebView.canGoBack()) {
04          mWebView.goBack();
05          return true;
06      }
07      return super.onKeyDown(keyCode, event);
08  }
```

在当前 Activity 处于界面最上方时，其内部的 onKeyDown()这个回调方法会在任意物理按键被单击时被触发。在这个方法中可以通过传入的 keyCode 参数来识别被按下的具体按键，代码第 03 行就是用于检测当前被按下的按钮是否是 Back 按钮，同时判断当前的 mWebView 是否存在可以返回到的页面，同时满足这两个条件时才执行 mWebView 的 goBack()方法并且返回 true 表示已消耗掉这个事件；如果不满足这两个条件则将事件传

递给下一级处理。如果不采取这种方式，则会导致该 Activity 不能够通过 Back 按钮退出的尴尬情况，读者可以自行测试一下。

至此，我们就实现了一个具备基本的浏览网页功能以及返回历史功能的“浏览器”，是不是特别的简单呢？由于 WebView 所涉及的相关内容非常多，例如如何与页面上的 JavaScript 脚本通信、如何个性化 WebView 等，因此，在本书第 10 章将更加详细地对 WebView 进行介绍，同时还将对 WebKit 内核进行简要的介绍。

8.6 Wi-Fi 的管理与使用

8.6.1 Wi-Fi 简介

8.1.3 节已经对 Wi-Fi 进行了较为详细的介绍。简而言之，Wi-Fi 就是一种可以将个人计算机、手持设备（如 PDA、手机）等终端以无线方式互相连接的技术。Wi-Fi 是英文“无线保真”的缩写，英文全称为 wireless fidelity，在无线局域网的范畴是指“无线相容性认证”，实质上是一种商业认证，同时也是一种无线联网的技术，以前通过网线连接计算机，而现在则是通过无线电波来连接。生活中比较常见的就是无线路由器，一旦一个无线路由器开始正常工作，那么在这个无线路由器的电波覆盖的有效范围都可以采用 Wi-Fi 连接方式连接到该路由器。如果无线路由器连接了一条 ADSL 线路或者别的上网线路，则又被称为“热点”。

随着用户对移动设备上网速度的要求越来越高，普通的 GPRS 上网的连接速度已经不能够满足人们的需求，然而相对高速的 3G 连接在国内目前还存在费率较高的问题，因此用户数量较少，而随着 Wi-Fi 的覆盖率越来越高，部分城市甚至能够实现市区全范围的 Wi-Fi 覆盖，并且多数免费，在一些人们相对长时间停留的场所例如机场、快餐店、咖啡厅等也通常会提供免费的 Wi-Fi 接入功能，对于个人用户也能够很容易地在自己的家中或办公场所使用无线路由器搭建起支持 Wi-Fi 的无线网络，在这样的背景下，移动设备上的 Wi-Fi 功能可以说是不可或缺的一部分，Wi-Fi 通信模块的低成本也使得目前大部分的移动设备都默认搭载了 Wi-Fi 模块。相信在不久的将来，在几乎所有的移动通信设备上都能够通过Wi-Fi的方式来连接到互联网。

8.6.2 Wi-Fi API

Android SDK 中的 android.net.wifi 包提供了控制和操作 Wi-Fi 设备所需要的 API，开发者在开发自己的应用程序时可以通过这些类来使用系统所可以识别到的 Wi-Fi 模块。这些 API 使得应用程序能够方便地对 Wi-Fi 进行管理，通过这些 API 还可以获得那些能够被搜索到的无线网络接入点的相关信息，包括接入点的网络连接速度、IP 地址、所处的地域等。其中还有一些类和方法提供了搜寻 Wi-Fi 网络、添加接入点、保存接入点、终止连接以及初始化连接的功能。

当然，这些类能够正常工作的前提条件是 Android 设备拥有可以使用的 Wi-Fi 无线通信模块，否则使用这些类将会导致抛出异常而造成程序的崩溃，因此在使用的时候应当确保

程序的健壮性，例如在需要使用 Wi-Fi 之前先判断系统是否支持 Wi-Fi，如果不支持则换用其他的方式例如 GPRS、EDGE、蓝牙等方式，不仅仅是对于 Wi-Fi，对于其他所有类似的情况都应当保持这个良好的习惯。

表 8-2 描述了 Android SDK 提供的用于管理 Wi-Fi 的类的作用说明。

表 8-2 Android Wi-Fi 相关类

类	说 明
ScanResult	用于描述一个已经被检测到的 Wi-Fi 接入点
WifiConfiguration	该类代表了一个已经配置好的 Wi-Fi 网络，包括了该网络的一些安全设置。例如接入点密码，接入点通信所采用的安全标准
WifiConfiguration. AuthAlgorithm	公认的 IEEE 802.11 标准认证算法
WifiConfiguration. GroupCipher	公认的组密码
WifiConfiguration. KeyMgmt	公认的密钥管理方案
WifiConfiguration. PairwiseCipher	公认的用于 WPA 的成对密码标准
WifiConfiguration. Protocol	公认的安全协议
WifiConfiguration. Status	网络所可能存在的状态
WifiInfo	描述了各个 Wi-Fi 连接的状态，该连接是否处于活动状态或者是否处于识别过程中
WifiManager	重要。它提供了用于管理 Wi-Fi 连接的各种主要 API
WifiManager. MulticastLock	允许应用程序接收 Wi-Fi 的多播数据包
WifiManager. WifiLock	允许应用程序永久地保持 Wi-Fi 连接(防止系统自动回收)

要管理 Wi-Fi 通常是使用 WifiManager 类作为切入点，WifiManager 类提供了用于管理 Wi-Fi 连接的一些主要 API，通过 Context. getSystemService(Context. WIFI_SERVICE) 方法可以得到它的一个实例。WifiManager 主要用于处理下面几类对象相关的事务：

- 已经配置好的网络连接列表。这个列表可以被用户查看或者更新，而且可以通过它来修改接入点的属性。
- 如果当前有连接存在的话，可以得到当前正处于活动状态的 Wi-Fi 连接的控制权，可以通过它建立或者断开连接，并且可以查询该网络连接的动态信息。
- 通过对已经扫描到的接入点的足够信息来进行判断，得出一个最好的接入点进行连接。
- 定义了很多用于系统广播通知的常量，它们代表了 Wi-Fi 状态的改变。

在这里顺便说一下 Android 对网络连接管理的方法，上面提到的 android. net. Wi-Fi 这些 API 是用于专门操作 Wi-Fi 连接的。如果对 Android 设备的抽象意义上的“网络连接”进行操作的话，可以使用 android. net. ConnectivityManager 这个类，ConnectivityManager 将会对应用程序发出的网络连接状态查询做出应答。当网络连接发生改变时，它还能通知各个应用程序。可以通过 Context. getSystemService(Context. CONNECTIVITY_SERVICE)方法来获取该类的一个实例。ConnectivityManager 所完成的主要任务是：

- 监控网络连接(包括 Wi-Fi、GPRS、UMTS 等)。
- 当网络连接发生改变时，向系统广播这一改变。
- 当失去了当前的网络连接时，尝试切换到另外一个连接。

• 提供了允许其他应用程序调用的API让应用程序可以方便地查询当前的网络状态。

要使用这些类所提供的方法，请读者详细地阅读Android的API文档，上面有详尽的说明。一般地，应用程序并不需要去对Wi-Fi连接进行操作，而是只需要通过这些类获取一些常用的信息如IP地址、MAC地址等，因为在Android默认的设置程序中已经能够很好地对Wi-Fi进行管理了。

8.7 Bluetooth的管理与使用

8.7.1 Bluetooth简介

8.1.4节已经对蓝牙的历史、协议规范以及优缺点等进行了较为详细的介绍，这里再做一些具体技术细节上的补充。蓝牙是一种支持设备短距离通信(一般10m内)的无线电技术，能在移动电话、PDA、无线耳机、笔记本计算机、相关外设等众多设备之间进行无线信息交换。蓝牙不仅仅是一项简单的技术，它更代表了一种崇尚简约和自由的信念，借助蓝牙，可以抛开传统连线的束缚，彻底地享受无拘无束的乐趣，现在，蓝牙已经被广泛地引入到移动电话和个人计算机上，使得用户完全摆脱了那些令人讨厌的连接电缆而可以直接通过蓝牙建立起无线通信连接。另外，如打印机、PDA、桌上型计算机、传真机、键盘、游戏操纵杆、耳机等其他的数字设备都可以成为蓝牙系统的一部分。蓝牙工作在全球通用的2.4GHz ISM(即工业、科学、医学)频段。蓝牙的数据速率为1Mbps。使用时分双工传输方案来实现全双工传输。蓝牙遵循的是IEEE 802.15协议。

ISM频带是对所有无线电系统都开放的频带，因此使用其中的某个频段都会遇到不可预测的干扰源，例如某些家电、无绳电话、汽车房开门器、微波炉等。为此，蓝牙特别设计了快速确认和跳频方案以确保链路稳定。跳频技术是把频带分成若干个跳频信道(hop channel)，在一次连接中，无线电收发器按一定的码序列(即一定的规律，技术上叫做“伪随机码”，就是“假”的随机码)不断地从一个信道“跳”到另一个信道，只有收发双方是按这个规律进行通信的，而其他的干扰不可能按同样的规律进行干扰；跳频的瞬时带宽是很窄的，但通过扩展频谱技术使这个窄带宽成百倍地扩展成宽频带，使得干扰的影响变得很小。

8.1.4节已经介绍过，蓝牙协议的核心协议层包括基带、链路管理协议(LMP)、逻辑链路控制和适配协议层(L2CAP)、服务发现协议(SDP)和无线射频通信(RFCOMM)。其中，基带层定义了蓝牙设备相互通信过程中必需的编解码、跳频频率的生成和选择等技术。LMP的作用主要是完成基带连接的建立和管理。L2CAP提供分割和重组业务。RFCOMM是用于传统串行端口应用的电缆替换协议。业务搜索协议(SDP)包括一个客户/服务器架构，负责侦测或通报其他蓝牙设备。关于蓝牙的最新信息，可以到蓝牙的官方网站上获取，蓝牙官方网站地址为http://www.bluetooth.com。

蓝牙作为一个全球公开的无线应用标准，通过把各种语音和数据设备用无线链路连接起来，使人们能够随时随地进行数据信息的交换和传输，这极大地方便和满足了广大人群的需求。Android在最初版本中是并不支持蓝牙设备的，不过随着系统的逐步完善，从Android 2.0版本才开始添加了对蓝牙的支持。

由于Android在2.0版本之后才提供了对蓝牙的支持，因此在建立Android项目时一

定要确保项目所使用的 Android 版本为 2.0 或更高，按目前的用户情况来看，推荐使用 2.2 版本进行开发，也可以使用更高的 2.3～4.0 版本，4.0 版本在用户界面做出了进一步的美化，本书的截图大部分是在 4.0 版本的模拟器中截取的。

8.7.2 Bluetooth API

Android SDK 中的 android. bluetooth 包提供了用于管理和使用蓝牙功能的类，通过这些类可以完成搜索蓝牙设备、连接蓝牙设备和通过蓝牙传输数据的功能。

具体来说主要是为应用程序提供了如下几个功能：

- 搜寻有效范围内的蓝牙设备。
- 通过本地的蓝牙适配器来查询到与之配对的蓝牙设备。
- 在配对的蓝牙设备之间建立 RFCOMM 信道。
- 连接到其他设备的指定端口。
- 在设备之间传输数据。

在 Android 应用程序中，如果需要用到蓝牙设备进行通信，在应用程序的配置文件 AndroidManifest. xml 必须声明权限，通过

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

来声明。如果需要用到一些其他特定的功能，例如请求蓝牙设备允许被搜索，还需要声明 BLUETOOTH_ADMIN 权限。通过添加

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

语句声明。另外需要注意的是，并不是所有运行 Android 系统的移动设备都保证提供了 bluetooth 硬件支持，因此在应用程序中也要考虑到这一点，防止程序在运行时崩溃。

表 8-3 描述了 SDK 提供的用于管理 Bluetooth 的接口和类的作用说明。

表 8-3 Android 提供的使用 Bluetooth 的接口和类

接口	描述
BluetoothProfile	蓝牙规范的公用 API 接口，所有的蓝牙规范都必须实现这个接口。Profile 目的是要确保 Bluetooth 设备间的互通性
BluetoothProfile. ServiceListener	用于在蓝牙客户设备连接或者断开连接时给它们发出通知的接口
类名	描述
BluetoothA2dp	该类作为对 BluetoothProfile 接口实现的实例，这是对蓝牙的 A2DP 规范的 API 实现类
BluetoothAdapter	代表了本地的蓝牙适配器
BluetoothAssignedNumbers	蓝牙的指令编号
BluetoothClass	代表了一个蓝牙的类，这个类描述了蓝牙设备的特征和性能参数
BluetoothClass. Device	定义了所有的 device 类所用的常量
BluetoothClass. Device. Major	定义了所有主要的 device 类所用的常量

续表

接　口	描　述
BluetoothClass. Service	定义了所有的 service 类所用的常量
BluetoothDevice	代表一个远程的蓝牙设备
BluetoothHeadset	实现蓝牙耳机服务的公共 API
BluetoothServerSocket	用于监听 Socket 连接请求的类
BluetoothSocket	一个已连接的或正在连接的 Socket 类

通过这些类应用程序可以对蓝牙进行控制和操作，例如打开或者关闭蓝牙，连接和断开其他的蓝牙客户设备，如蓝牙耳机等，还可以与其他设备之间建立起 RFCOMM 协议的连接并且传输文件。

8.7.3 Bluetooth 示例

1. 搜寻可连接到的蓝牙设备

该示例说明了如何开启和关闭蓝牙设备，如何使得本机蓝牙设备可以被其他蓝牙搜寻到，以及如何执行一次搜寻工作，相关截图如图 8-26 所示。



图 8-26　蓝牙搜寻示例

如图 8-26 所示，左边两个截图分别是在蓝牙关闭和开启的状态下截取的，两张截图的区别是在上方状态栏，可以发现第二张截图的状态中多出了一个蓝牙的标志。第三张截图是在单击了“允许搜索”按钮后截取的，对话框的内容是让用户选择是否允许应用程序请求蓝牙可被搜寻到，这个对话框是由 Android 系统提供的，每当有代码发出使得蓝牙可被搜寻的请求时，这个对话框就会被呼出。第四张截图是在单击“开始搜索”按钮后截取的，截图中正处于搜寻蓝牙设备的状态，可以看到在后方的搜寻结果列表中已经出现了两个设备，分别列出了设备的硬件地址和设备名称。

1）权限声明

要使用系统的蓝牙设备，应用程序必须在 AndroidManifest. xml 文件中做出如下声明：

```
01 <!-- SDK 的版本至少要高于 5 -->
02 <uses-sdk android:minSdkVersion="5" />
03 <!-- 声明需要使用蓝牙的权限 -->
04 <uses-permission android:name="android.permission.BLUETOOTH" />
05 <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

第 02 行的声明是为了确保已安装的 Android 版本支持对蓝牙设备的访问，第 04 和第 05 行则是用于申请对蓝牙设备的访问权限。

2）开启和关闭蓝牙设备

要开启或者关闭蓝牙设备，可以直接通过 BluetoothAdapter 提供的 enable()和 disable()方法，在本例中将如下两个方法与相应的按钮进行绑定，从而实现了通过按钮来开启蓝牙的功能。

```
//开启蓝牙设备
public void enableBluetoothDevice(View view)
{
    bluetoothAdapter.enable();
}

//关闭蓝牙设备
public void disableBluetoothDevice(View view)
{
    bluetoothAdapter.disable();
}
```

3）使蓝牙可被搜寻

蓝牙设备是否能够被其他设备搜寻到，并不仅仅取决于是否开启了蓝牙设备，还要取决于这个蓝牙设备是否允许被其他设备发现，这是出于对蓝牙安全性的考虑，因此，如果需要使得蓝牙设备能够被其他蓝牙发现，需要首先开启蓝牙的可见性，代码如下：

```
//使设备可以被其他蓝牙设备搜索
public void makeBluetoothDeviceDiscoverable(View view)
{
    Intent enableDiscovery = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
    startActivityForResult(enableDiscovery, REQUEST_DISCOVERABLE);
}
```

4）搜寻有效范围内可连接的设备

使用 Bluetooth API 可以直接开始对其他蓝牙设备的搜寻，即使用：

```
bluetoothAdapter.startDiscovery();
```

方法，该方法将启动一次搜寻过程，并且有一个默认的搜寻超时时间，使得搜寻任务在超过该时间后自动停止搜索，在每次搜寻任务的过程中，将会产生如下两类有价值的消息(action)，即：

- 发现一个蓝牙设备时产生的消息(BluetoothDevice.ACTION_FOUND)。
- 搜寻结束时产生的消息(BluetoothAdapter.ACTION_DISCOVERY_FINISHED)。

为了实时地将搜寻到的设备显示出来，可以利用上述的第一类消息，为此，实现一个用于处理这类消息的 BroadcastReceiver，专用于在 BluetoothDevice. ACTION_FOUND 消息产生时做出相应的处理：

```
//该接收器会接收到"搜索过程中发现一个蓝牙设备"的消息,然后做出处理
private BroadcastReceiver foundEventReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        //获取搜索到的设备信息
        BluetoothDevice device = intent
                .getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        //添加搜索到的设备信息
        attachableDevices.add(device);
        //更新已搜索到的设备列表
        showDevices();
    }
};
```

为了使得 BluetoothDevice. ACTION_FOUND 事件的发生能够被上面实现的接收器所接收，需要在代码中为接收器绑定消息过滤器并且注册到系统中（通常在 onCreate()方法中加入这段代码）：

```
//注册广播接收器
IntentFilter foundEventFilter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(foundEventReceiver, foundEventFilter);
```

这样，每当 BluetoothDevice. ACTION_FOUND 事件发生时，foundEventReceiver 的 onReceive 方法将被触发，从而将新发现的设备添加到设备列表，并且更新显示，将新发现的设备信息显示出来，更新显示的方法 showDevices()代码如下：

```
//显示搜索到的设备列表,在搜索过程中更新
protected void showDevices()
{
    List<String> bluetoothDeviceList = new ArrayList<String>();
    for (int i = 0, size = attachableDevices.size(); i < size; ++i)
    {
        BluetoothDevice bluetoothDevice = attachableDevices.get(i);
        String bluetoothDeviceInfo = bluetoothDevice.getAddress()
                + "\n" + bluetoothDevice.getName();
        bluetoothDeviceList.add(bluetoothDeviceInfo);
    }

    final ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, bluetoothDeviceList);
    mHandler.post(new Runnable() {
        public void run()
        {
            setListAdapter(adapter);
        }
    });
}
```

showDevice()方法根据目前 attachableDevice 列表中的存在的 BluetoothDevice 对象，依次获取需要的信息并保存到字符串列表中(代码第 04～11 行)，然后显示到界面中去(代码第 13～20 行)。

5）结束搜寻

良好的编程习惯要求我们在对资源使用之后释放该资源，因此，应该在搜寻完成后注销掉上一步中注册的接收器，为了实现这个目的，可以通过对 BluetoothAdapter. ACTION_DISCOVERY_FINISHED 事件的监听获取到“搜寻完成”这个事件，与前面相似，可以通过注册另一个接收器来实现：

```
//注册广播接收器
IntentFilter discoveryFilter = new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
registerReceiver(discoveryEndEventReceiver, discoveryFilter);
```

上面注册的 discoveryEndEventReceiver 的代码如下：

```
01  //该接收器会接收到"搜索过程结束"的消息,然后做出处理
02  private BroadcastReceiver discoveryEndEventReceiver = new BroadcastReceiver() {
03
04      @Override
05      public void onReceive(Context context, Intent intent)
06      {
07          //注销消息接收器
08          unregisterReceiver(foundEventReceiver);
09          unregisterReceiver(this);
10          //搜寻完成标志
11          discoveryFinished = true;
12      }
13  };
```

这样，当 BluetoothAdapter. ACTION_DISCOVERY_FINISHED 时间发生时，这个监听器将在代码中注销消息接收器并且将搜寻完成标志置为 true(代码第 08 行、第 09 行和第 11 行)。

2. 使用蓝牙传输数据

在本节第 1 部分获取了可供连接配对的蓝牙设备列表之后，就可以通过一定的方法来建立起与远端蓝牙设备的连接，然后通过这个建立好的连接在传输数据。两个蓝牙设备之间的通信类似于前面介绍的 Socket 通信，相应的两个类为 BluetoothServerSocket 和 BluetoothSocket，它们的用法与 ServerSocket 和 Socket 类基本类似，因此这里仅简单地对其进行说明，读者可以结合 SDK 提供的 samples 中的 BluetoothChat 示例来进行学习，本节中的部分代码段就是截取自 BluetoothChat。

BluetoothChat 实现的功能是利用蓝牙连接进行即时聊天，它包括了 3 个类，分别是：

- BluetoothChat——提供了聊天的主界面，并且可以通过 menu 按键呼出菜单，从而进行“使本机蓝牙可被发现”和“连接到聊天对象设备”的操作。
- BluetoothChatService——这个类的作用是配置好两台设备之间的蓝牙连接并且对

建立好的连接进行管理。这个类中包括了 3 个主要的线程类：AcceptThread——用于等待另一端（聊天的对方）的连接并且在接收到连接请求时创建连接；ConnectThread——用于向另一端发出连接请求，如果另一端正常工作则能够建立起连接；ConnectedThread——这个线程将在连接成功建立之后保持运行，并且负责处理所有接收数据及发送数据的操作。

- DeviceListActivity——这个类的作用就跟本节第 1 部分实现的功能类似，即搜寻附近可连接到的蓝牙设备，不同的地方是该类实现了通过点击列表中的项来进行连接的功能。

下面一起来看一下这个示例中关键的 3 个线程类的代码，这里先要说明一点是：在一次连接建立完成的过程中，发生连接的两端并不会同时使用这三个线程，因为这两端在建立连接的过程中所扮演的角色并不相同，发起连接请求的一段所扮演的是客户端的角色，而接收到连接请求的一段所扮演的则是服务端的角色。作为服务端的一方要使用到的是 AcceptThread 和 ConnectedThread，作为客户端的一方要使用到的是 ConnectThread 和 ConnectedThread。首先来看 AcceptThread 线程的部分代码：

```
private class AcceptThread extends Thread {
    //本地的服务监听端口
    private final BluetoothServerSocket mmServerSocket;
    BluetoothSocket socket = null;
    public AcceptThread() {
        BluetoothServerSocket tmp = null;
        // 创建一个新的监听端口
        try {
            tmp = mAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
        } catch (IOException e) {}
        mmServerSocket = tmp;
    }
    public void run() {
        //等待连接，阻塞线程，直到连接建立或者异常退出
        while (mState != STATE_CONNECTED) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                break;
            }

            // 接收到了连接请求
            if (socket != null) {
                synchronized (BluetoothTranService.this) {
                    switch (mState) {
                    case STATE_LISTEN:
                    case STATE_CONNECTING:
                        // 状态正常，已连接
                        connected(socket, socket.getRemoteDevice());
                        break;
                    case STATE_NONE:
```

```
32                  case STATE_CONNECTED:
33                      // 状态异常,关闭连接
34                      try {
35                          socket.close();
36                      } catch (IOException e) {}
37                      break;
38                  }
39              }
40          }
41      }
42  }
43
44  public void cancel() {
45      try {
46          mmServerSocket.close();
47      } catch (IOException e) {}
48  }
49 }
```

该线程类将在本机作为服务端参与连接的建立时被使用到，线程开始执行后可以通过 listenUsingRfcommWithServiceRecord(String，UUID)方法得到一个 BluetoothServerSocket 对象（代码第 09 行），方法中 String 类型的参数代表了本机的名称，UUID 是用于使用蓝牙设备的应用程序之间相互识别的一个唯一识别码，当这个 UUID 在客户端和服务端是同一个值时才能够建立起连接，因此如果所开发的应用需要使用蓝牙通信，则需要这样一个 UUID 来达到配对的作用。然后通过 mServerSocket 的 accept()方法开始监听连接到这个端口的请求（代码第 17 行）。该监听线程会一直阻塞直到有新的请求到来，除非在程序中人为地调用 mServerSocket 的 close()方法，当接收到连接请求后，再根据当前的连接状态来执行相应的操作（代码第 23～40 行），通常是执行第 29 行代码，即连接正常，开始执行 ConnectedThread 线程来进行聊天。

反之，当本机作为客户端参与连接的建立时，将会使用到 ConnectThread 线程，这个线程的部分代码如下：

```
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        mmDevice = device;
        BluetoothSocket tmp = null;

        // 根据需要连接到的 device 获得一个 BluetoothSocket 对象,用于建立连接
        try {
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) {}
        mmSocket = tmp;
    }
```

```
15
16      public void run() {
17          //由于已经决定并开始了连接过程,因此停止搜寻的过程,减少资源占用
18          mAdapter.cancelDiscovery();
19
20          try {
21              //请求连接,阻塞线程,直到连接建立或者异常退出
22              mmSocket.connect();
23          } catch (IOException e) {
24              connectionFailed();
25              try {
26                  mmSocket.close();
27              } catch (IOException e2) {}
28              //建立连接失败,重新开始
29              BluetoothChatService.this.start();
30              return;
31          }
32
33          //线程任务完毕,销毁线程
34          synchronized (BluetoothChatService.this) {
35              mConnectThread = null;
36          }
37
38          //状态正常,已连接
39          connected(mmSocket, mmDevice);
40      }
41
42      public void cancel() {
43          try {
44              mmSocket.close();
45          } catch (IOException e) {}
46      }
47  }
```

如上面的代码所示,首先通过 BluetoothDevice 类的方法来得到 BluetoothSocket(代码第 11 行),可以得到一个用于连接到远程蓝牙设备的 BluetoothSocket 对象,该方法的参数 MY_UUID 必须和服务端的 UUID 是同一个数值,否则不能够建立起连接。得到了 BluetoothSocket 对象后,通过调用它的 connect()方法(代码第 22 行),如果一切正常将建立起到另一端的一条专用的蓝牙连接供聊天使用,如果出现异常则重新开始连接过程(代码第 29 行)。

通过前面两个线程建立起连接之后,即可以使用 ConnectedThread 线程来进行即时聊天,在这个方法中首先从建立好的 Socket 连接处获取输入和输出流,然后就可以使用这两个流进行读取消息和发送消息的操作,实现过程比较简单,部分代码如下:

```
private class ConnectedThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final InputStream mmInStream;
```

```
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        mmSocket = socket;
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // 从 BluetoothSocket 处获取输入流和输出流
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) {
            Log.e(TAG, "temp sockets not created", e);
        }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    public void run() {
        byte[] buffer = new byte[1024];
        int bytes;

        //在 while 循环体中监听另一端发送过来的聊天内容,并将聊天内容显示到界面
        while (true) {
            try {
                bytes = mmInStream.read(buffer);
                mHandler.obtainMessage(BluetoothChat.MESSAGE_READ, bytes, -1, buffer)
                        .sendToTarget();
            } catch (IOException e) {
                connectionLost();
                break;
            }
        }
    }

    //发送聊天消息,并且也将消息显示到自己的聊天界面
    public void write(byte[] buffer) {
        try {
            mmOutStream.write(buffer);
            mHandler.obtainMessage(BluetoothChat.MESSAGE_WRITE, -1, -1, buffer)
                    .sendToTarget();
        } catch (IOException e) {}
    }

    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) {}
    }
}
```

本示例的流程如图 8-27 所示。

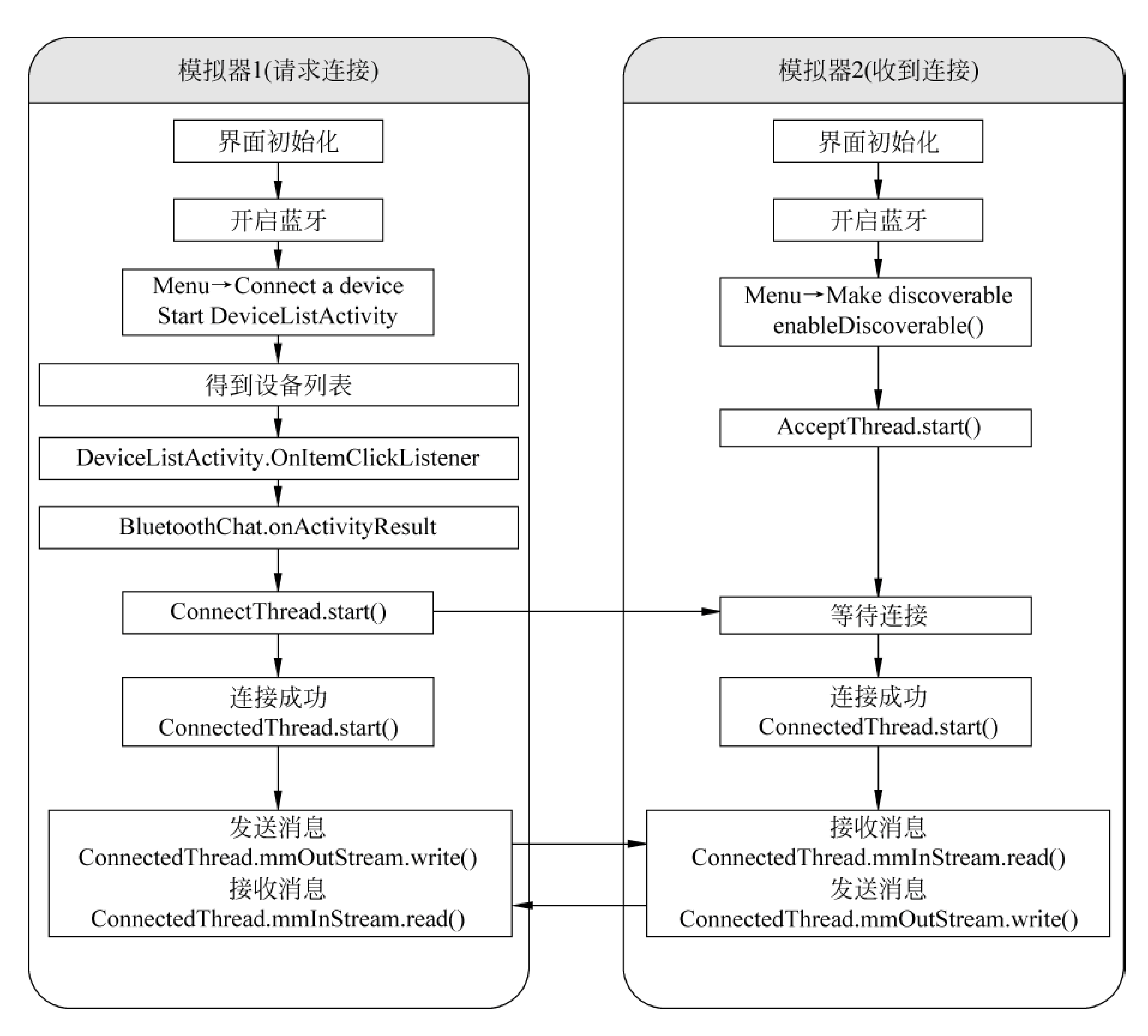


图 8-27 蓝牙聊天应用流程图

本示例实现了通过蓝牙聊天的功能，将此示例的代码稍作改变，可以使得两个设备之间能够通过蓝牙传输文件，请读者结合 8.3 节的内容实现蓝牙传输文件的功能。

8.8 NFC

8.8.1 NFC 简介

8.1.5 节已经对 NFC 进行了较为详细的介绍。与蓝牙一样，在新版本的 Android 中，近场通信才正式得到支持。

NFC 与蓝牙相比较，它不需要复杂设置程序，也可以简化蓝牙连接。NFC 略胜蓝牙的地方在于设置程序较短，但是无法达到蓝牙的低功率。其最大数据传输量是 424kbps 远小于蓝牙 v2.1 的 2.1Mbps。虽然 NFC 在传输速度与距离方面比不上蓝牙，但是由于 NFC 技术不需要电源，对于移动电话和消费性电子产品来说，这种技术的使用比较方便。它的短距离通信特征正是其优点，由于耗电量低、一次只和一台机器连接，因此其拥有较高的保密

性与安全性，例如 NFC 技术可以在使用信用卡交易时避免被盗用。NFC 的目标并非是取代蓝牙等其他无线技术，而是在不同的场合、不同的领域起到相互补充的作用。

虽然 Android 系统已经提供了关于 NFC 的 API，但是与蓝牙和 Wi-Fi 一样，它必须有硬件的支持。在实际硬件的支持上面，现在的情况还不是很普及，前段时间有新闻称 Google 将在纽约和旧金山测试 NFC 支付，而 iPhone 4S 还是不会集成 NFC 的功能，Android 方面，Google 早在 Nexus S 手机上就已经内置了 NFC。

NFC 通信总是涉及一个发起设备和一个目标设备，可以理解为一个阅读器和一个标签设备，阅读器可以产生一个射频场用来给一个标签提供电源，正是这个类似于 RFID 的特性使得一个 NFC 的标签设备可以做成是一个简单的标签、卡片的无源的形式。另外，当通信的双方都开启了阅读器的情况下，也可以通过 RFC 建立起点到点的通信。

一个拥有 NFC 硬件支持的 Android 设备典型的设置是在屏幕未锁的状态下工作在 NFC 通信的发起设备模式，这个模式就是通常所说的标签读写器。工作在这个模式下的 Android 设备将会主动地去搜寻有效范围内的 NFC 标签，并且在适当的时候对这些搜寻到的标签进行处理工作。Android 2.3.3 版本下还加入了少量的对 NFC 的 P2P 方面的支持。

NFC 的标签有很多种不同的种类，包括了各种复杂程度，随着复杂程度的不同，它们所存储的信息量及种类也有所不同，例如某些简单的标签仅仅提供了供读写的语义，并且只提供了小块一次性写入的只读存储；而一些稍微复杂一点的标签还提供了一些数学运算的功能，支持加密从而可以进行身份验证功能；最复杂的标签上面还可以包括操作环境，允许在标签上面执行代码并且进行复杂的交互。

8.8.2　NFC API

本节具体介绍 Android 为 NFC 提供的 API。

Android 中提供的与 NFC 相关的较高层的类都包含在 android.nfc 中，这个包中包含了用于与本地 NFC 适配器交互的类，用于代表已经识别的标签的类以及用来使用 NDEF (NFC Data Exchange Format)格式的类。android.nfc 中的每个类的具体用途如表 8-4 所示。

表 8-4　NFC API

类　名	描　述
NfcManager	NFC 的一个高级管理类，用于枚举出本机的 NFC 适配器。由于大多数的设备都只提供一个 NFC 适配器，因此在大多数情况下我们可以通过 getDefaultAdapter(Context)这个静态的方法来得到本地的 NFC 适配器的引用
NfcAdapter	该类代表了本地的 NFC 适配器。它定义了如何将 NFC 标签的信息传达给 Activity 的 intent，并且提供了用于注册前台标签调度和前台的数据推送的方法。前台的基于 NDEF 的数据推送是目前 Android 仅提供的点到点支持方式

续表

类　名	描　述
NdefMessage and NdefRecord	NDEF 是由 NFC 论坛所定义的一种数据结构，它是为了高效率地在 NFC 标签上存储数据，例如文本、url 或者其他的数据格式。NdefMessage 是用于封装需要传输或读取的数据的容器。而每一个 NdefMessage 包含了 0 个或若干个 NdefRecord。每个 NDEF 的记录都包括一种有效类型的数据。在一条 NDEF 消息中的第一个记录的作用是向 android 的 Activity 调度一个标签
Tag	该类代表了一个无源的 NFC 标签。这种标签可以来自于各种物体，比如通常所用的物理标签、卡片、钥匙卡或者也可以是一部可以仿真成 NFC 标签的电话机。当 NFC 识别设备发现了一个 NFC 标签，系统就会创建一个 NFC Tag 对象并在对象中封装一个 intent。然后 NFC 的标签调度系统将会负责将这些 intent 调度到适当的 Activity 中。可以通过 getTechList()方法来决定使用适合该 Tag 对象的读写标准并且可以使用 android. nfc. tech 包所提供的相关类来创建与此相关的 TagTechnology 对象

android. nfc. tech 包则主要是包括了一些用于查询特定标签的特性和 I/O 操作的类。所有的这些类都需要实现 TagTechnology 接口，包括 NfcA、NfcB、NfcF、NfcV、IsoDep、Ndef、NdefFormtable、MifareClassic、MifareUltralight 这几个类。

8.8.3 NFC 示例

8.8.2 节简单介绍了与 NFC 编程相关的类，下面介绍一下如何进行 NFC 的编程。由于 NFC 编程需要支持 NFC 功能的真机，而考虑到这种真机还不是很普及，因此 Android SDK 为我们提供了一个示例——NFCDemo，这个示例模拟出了一个虚拟的 NFC Tag，从而对 NFC 工作的原理进行演示，本节将通过这个示例来对 NFC 编程进行介绍。

1. AndroidManifest 文件

根据 Android 系统所遵循的权限机制，需要在 AndroidManifest. xml 中对需要使用的权限进行声明，这样在程序进行安装时将会提示用户该程序将会使用到哪些设备，让用户来决定是否赋予应用程序相应的权限，Android 通过这样的机制来提高系统的安全性。因此当我们在应用程序中需要使用到 NFC 硬件的时候，应当在 AndroidManifest. xml 中加上如下权限声明：

```
<uses - permission android:name = "android.permission.NFC" />
```

另外还需要声明的是能够支持应用程序的最低 SDK 版本，由于对 NFC 的支持是在 API 10 以后才比较完善，因此需要添加：

```
<uses - sdk android:minSdkVersion = "10"/>
```

一般来说，android 的应用程序都是通过上传到 Android Market 或着类似的应用商店

上供用户下载的，而这些应用商店通常会根据你拥有的机型来过滤出适合你的手机的应用，为了使得 NFC 应用程序能够很好地被归类，可以通过加入如下一段声明使应用程序能够被归类到“支持 NFC”的手机型号中：

```
<uses-feature android:name="android.hardware.nfc" android:required="true" />
```

此外，为了告知系统本应用程序能够处理 NFC 相关的事件，可以通过声明一些 intent filter 来告诉操作系统该 Activity 可以处理 NFC 数据，有 3 种声明的方式：

```
<intent-filter>
  <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
  <data android:mimeType="mime/type" />
</intent-filter>

<intent-filter>
  <action android:name="android.nfc.action.TECH_DISCOVERED"/>
  <meta-data android:name="android.nfc.action.TECH_DISCOVERED"
       android:resource="@xml/nfc_tech_filter.xml" />
</intent-filter>

<intent-filter>
  <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

这 3 种 intent filter 的声明方式分别适用于特定的情况：

- NDEF_DISCOVERED——当一个标识为 NDEF 的标签被扫描到时，系统将会产生一个包含这个 action 的 Intent，如果有应用程序能够处理这个 action，那么这个应用程序将被呼出并对其进行处理。这种 Intent 的优先级别最高，即当系统中存在能够处理这种事件的应用程序时，后面的两种 Intent 都不会产生。处理这种 Intent 的 Activity 一般在它的 intent-filter 中还会添加 data 类型的过滤器，可以认为这是一种相对更精确的标签匹配方式。
- TECH_DISCOVERED——如上所见，这种事件所产生的 Intent 通常附加地需要匹配 meta-data 中的条件，因为通常来说一个 Activity 会特定地用于处理某种技术的标签，这样可以使得对使用多种技术的标签能够合适地与应用程序进行匹配。这种 Intent 的优先级别低于 NDEF_DISCOVERED，高于 TAG_DISCOVERED。
- TAG_DISCOVERED——当有标签被扫描到时会发生这个 Intent，当然前提是没有前面介绍的两种匹配方式发生，这种 Intent 的优先级别最低，通常用于在没有特定的应用程序对标签进行处理时为其提供一种默认方式。

使用上面的何种匹配方式取决于具体的应用类型，因此需要通过分析具体的需求来决定使用哪一种方式。具体如何选择将在后面提到(即 8.8.3 节第 2 部分“标签调度系统”)。这里可以看到一个完整的 AndroidManifest.xml 例子(摘自 NFCDemo)。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.example.android.nfc" >

```

```
04      <uses-permission android:name="android.permission.NFC" />
05      <uses-permission android:name="android.permission.CALL_PHONE" />
06
07      <application
08          android:icon="@drawable/icon"
09          android:label="@string/app_name" >
10          <activity
11              android:name=".simulator.FakeTagsActivity"
12              android:theme="@android:style/Theme.NoTitleBar" >
13              <intent-filter >
14                  <action android:name="android.intent.action.MAIN" />
15                  <category android:name="android.intent.category.LAUNCHER" />
16              </intent-filter>
17          </activity>
18          <activity
19              android:name="TagViewer"
20              android:theme="@android:style/Theme.NoTitleBar" >
21              <intent-filter >
22                  <action android:name="android.nfc.action.TAG_DISCOVERED" />
23                  <category android:name="android.intent.category.DEFAULT" />
24              </intent-filter>
25          </activity>
26      </application>
27
28      <uses-sdk android:minSdkVersion="9" />
29      <uses-feature android:name="android.hardware.nfc" android:required="true" />
30
31  </manifest>
```

该 AndroidManifest.xml 文件中，第 04 行和第 05 行声明了该应用所需要的权限，其中包括了对需要使用 NFC 的权限声明，可以看到第 22 行是用于告诉操作系统该 Activity 可以处理“发现 NFC 标签”这个事件，采用了前面提到的 3 种方式中的最后一种方式。

2. 标签调度系统

在对示例的具体实现进行介绍之前，首先介绍一下 Android 的标签调度系统。简而言之，标签调度系统就是用于在 Android 系统获取到一个新的标签发现事件之后，决定如何对系统 Activity 进行调度从而用于对该事件进行处理的系统。

在一个 NFC 标签被扫描到之后，我们所期望设备进行的操作就是设备能够自主选择能够处理该 NFC 标签的最合适的 Activity 来对之进行处理。要求设备能够自主选择 Activity 而不是过于依靠用户的选择，这是由 NFC 技术的特征所决定的：设备通常需要非常靠近 NFC 标签才能进行扫描工作。因此如果对一个标签的扫描操作需要用户参与，通常表现为需要用户在屏幕上选择多个 Activity 中的一个，这个操作通常容易导致用户将设备从标签处移开从而使得这个 NFC 的连接断开，这显然不是我们希望看到的结果。

因此，在开发应用程序时，应当通过本节第 1 部分介绍的方式（在 AndroidManifest.xml 文件中设置 intent-filter）或者在代码中定义 IntentFilter 的方式，来使得应用程序只会去处理那些我们关心的 NFC 标签类型，从而减小 Activity 选择对话框的弹出几率。

为了更好地实现这种需求，Android 还提供了两类标签调度系统来帮助我们的应用程序能够准确地识别出应该处理的标签，它们分别是：

- 基于 Intent 的标签调度系统。
- 基于最前端 Activity 的标签调度系统。

基于 Intent 的标签调度系统的工作机制是：当新的标签被发现时，系统将会去检查所有 Activity 的 intent-filter，结合这些 Activity 所能够处理的数据类型来查找这些 Activity 中最适合的一个来处理 NFC 标签。如果在所有的 Activity 中有两个或以上的 Activity 以相同的 intent-filter 和数据类型来声明能够处理这种类型的标签，系统就会呼出一个 Activity 选择对话框请求用户在这几个 Activity 中进行选择。这种标签调度主要基于前面介绍的在 AndroidManifest.xml 文件中所声明的内容。

基于最前端 Activity 的标签调度系统的工作机制是：它拥有比基于 Intent 的标签调度系统更高的优先级，即当目前运行在最前端的 Activity 能够对该标签进行处理时，那么它将能够优先捕获并处理这个标签。这种标签调度系统能够工作的前提是当前处于最前端的 Activity 能够对标签进行处理，否则这个标签将会被发送给基于 Intent 的标签调度系统进行后续的处理。

这两种标签调度系统的使用方法请读者自行阅读 SDK 文档中 Dev Guide 下 Framework Topics 中的 Near Field Communication 主题。

3. NFCDemo

本部分来介绍一下 NFCDemo 这个来自于 SDK samples 中的示例，首先在 Eclipse 中导入该示例（New→Android Project→Create project from existing sample→Android 2.3.3（或以上）→NFCDemo），导入后可以发现该示例存在一些错误，即一些以 com.google.common 开头的类不能被找到，这是由于这些类是由 Google 另外的 Java 类库提供的，而这个示例中默认并没有包含这个类库，因此需要另外去下载，提供这个类库的项目名称为 Guava，项目在 Google Code 上的地址为：

```
http://code.google.com/p/guava-libraries/
```

在这里可以找到最新的 Guava 类库下载，例如目前最新版本是 guava-10.0.1.jar，如图 8-28 所示。

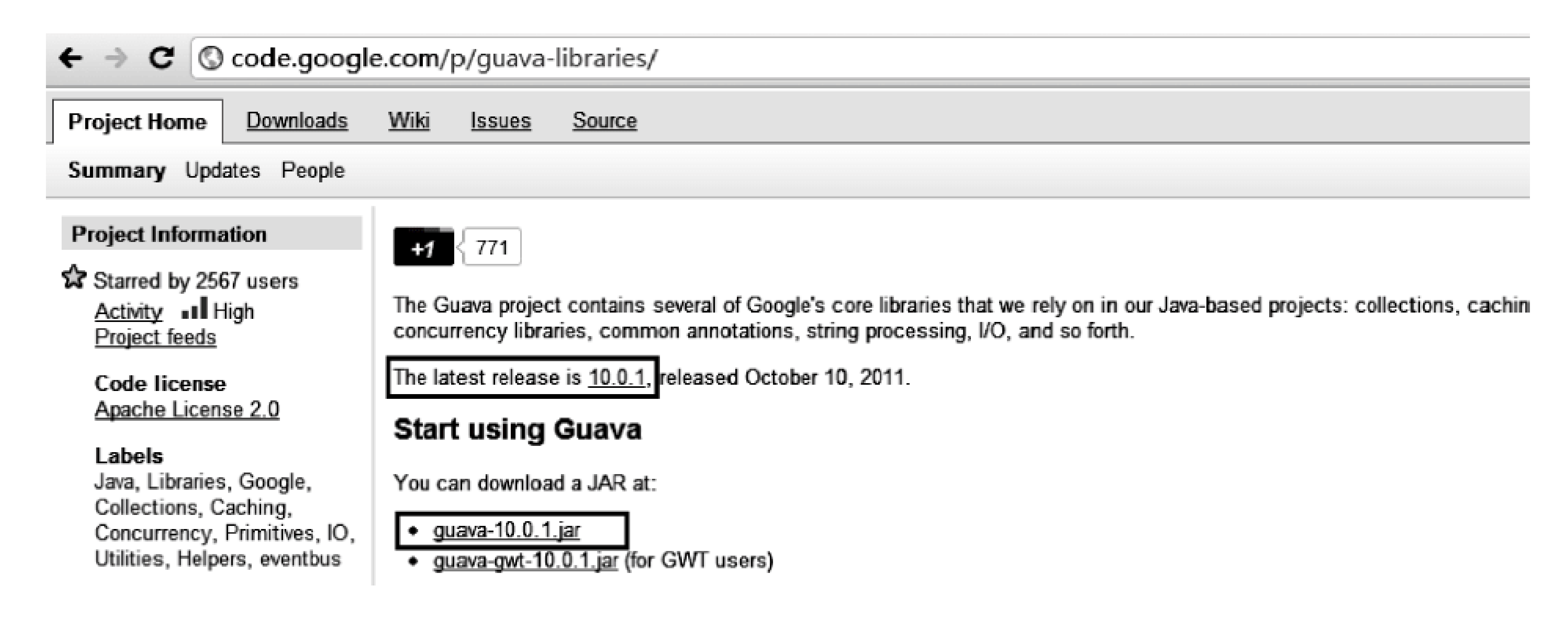


图 8-28　到 Guava 主页下载所需的包

下载得到 guava-10.0.1.jar 包之后，在 NFCDemo 下新建一个 lib 文件夹，然后将 guava 包复制到这个文件夹下，再通过右击 NFCDemo→Build Path→Configure Build Path→Libraries→Add JARs→NFCDemo→lib→guava-10.0.1.jar 的方式将该类库与项目关联起来，如果操作正确，则可以发现 NFCDemo 项目的错误已经解决了，所有的包都已经能够被找到。

1）示例效果

在前面增添了 guava 类库从而解决了 NFCDemo 的错误之后，首先运行该示例来观察一下示例的效果，如图 8-29 所示。

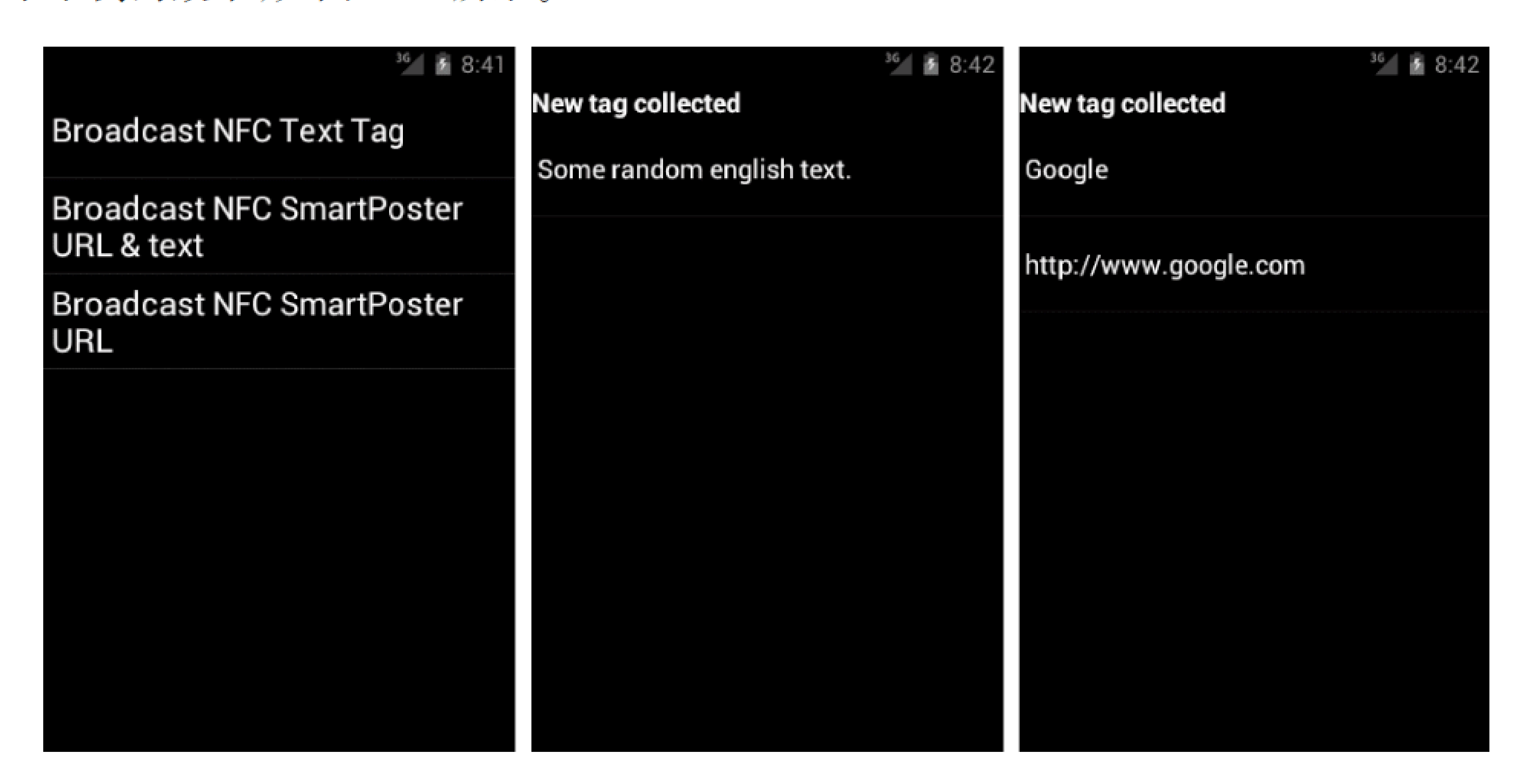


图 8-29 NFCDemo 运行效果

如图 8-29 所示，第一幅截图是 NFCDemo 启动后的界面，可以看到这个 Activity 呈现出来的是一个列表，分别点击这 3 个列表选项将会弹出新的界面，图 8-29 中给出了点击前面两个选项时得到的新界面，第二幅截图所体现的是系统接收到了一个带有纯文本消息的 NFC 标签，第三幅截图则是体现了系统接收到一个带有文本和网址两种信息的 NFC 标签。

2）代码解析

NFCDemo 代码包括 3 个包：

- com.example.android.nfc——这个包的作用是接收并处理 NFC 标签，将标签中的信息显示到界面中，包括了两个类：NdefMessageParser 类——用于解析标签携带的数据（该类实际上是通过迭代的方式调用了其他类的相关方法）；TagViewer 类（Activity）——用于显示 NFC 标签中包含的一些可读数据（同样是通过迭代的方式调用了其他类的相关方法）。
- com.example.android.nfc.record——这个包所包含的是一些代表具体的 NFC 标签中的不同数据记录的类，即 SmartPoster（一种由 NFC Forum 所维护的标签格式）、TextRecord（包含字符串的记录）和 UriRecord（包含 Uri 的记录），它们都实现了 ParsedNdefRecord 接口，这个接口包含一个 getView 方法，用于为上级调用者返回一个视图（View）对象，从而可以使得各种不同种类的 Record 能够按不同方式显示。

- com. example. android. nfc. simulator——这个包中包括了另一个 Activity，这个 FakeTagsActivity 用于发出伪造的 NFC 标签发现事件，从而使得我们可以在不用有真机和实体标签的情况下来体验 NFC 编程，另一个类 MockNdefMessages 中则定义了一系列的字节数组，用于模拟通过扫描实体 NFC 标签可以得到的标签中所包含的字节数据。

这里需要对 SmartPoster 这种格式进行简单介绍，SmartPoster 是由 NFC Forum 所定义的一种 Record 类型，这种 Record 类型可以理解为一种复合类型，它可以由 TextRecord 和 UriRecord 这两种类型的 Record 组合而成，其中 TextRecord 是可选的，即一个 SmartPoster 既可以包含一个或多个 TextRecord，也可以不包含 TextRecord，而要求必须包含一个并且只能有一个的 UriRecord。

NFCDemo 的运行过程的示意图如图 8-30 所示。

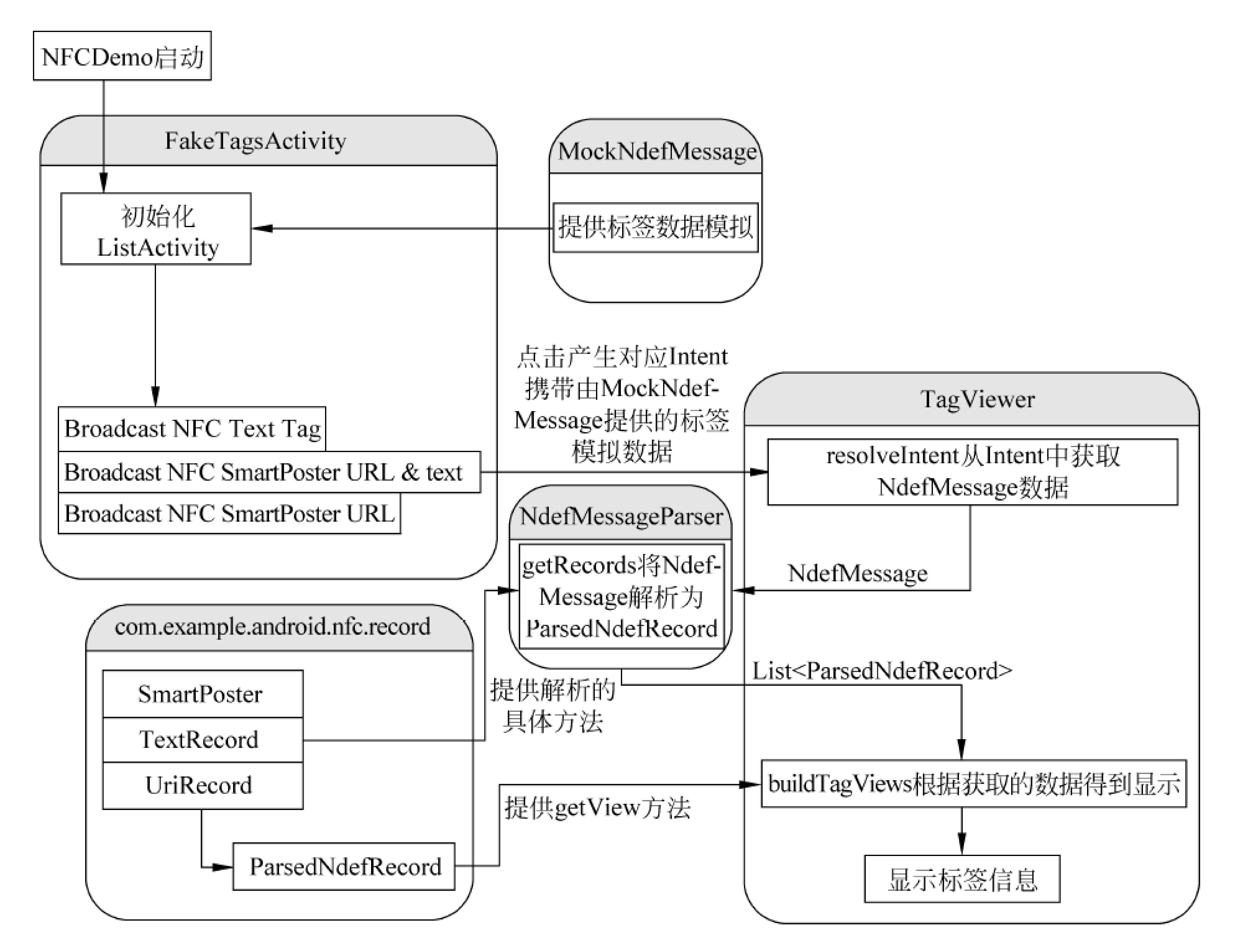


图 8-30 NFCDemo 运行过程示意图

读者结合前面提供的信息再去阅读 NFCDemo 的代码，就能够比较全面地对 NFC 应用的工作方式进行了解了。

3）为 NFCDemo 添加小功能

细心的读者应该已经发现，在 NFCDemo 的项目文件夹/res/raw 下存在着一个声音文件：discovered_tag_notification. ogg，然而在 NFCDemo 中似乎没有发现对它的使用，根据它的文件名可以大致猜测出这个音频文件是用于对"发现新标签"这个时间进行提醒的一个声音消息。在 NFCDemo 的运行过程中也没有听见什么声音，读者可以在 PC 的播放器中

播放一下这个音频文件，听一听它的声音。

虽然这个示例本身并没有声音提示的功能，但是可以很容易地来实现这个功能，参照7.1.1节的内容，可以在TagViewer的resolveIntent方法中buildTagViews方法之后插入如下代码(加粗部分)：

```
// Setup the views
setTitle(R.string.title_scanned_tag);
buildTagViews(msgs);
MediaPlayer mediaPlayer = MediaPlayer.create(this, R.raw.discovered_tag_notification);
mediaPlayer.start();
```

这样就可以实现在扫描到标签时发出提示音了。

参考文献

1. 维基百科“GSM”词条：zh.m.wikipedia.org/wiki/GSM.
2. 百度百科“GPRS”词条：baike.baidu.com/view/1307.htm.
3. MBA智库百科“EDGE”词条：http://wiki.mbalib.com/wiki/EDGE.
4. 维基百科“3G”词条：http://zh.wikipedia.org/wiki/3G.
5. 维基百科“Wi-Fi”词条：http://en.wikipedia.org/wiki/Wi-Fi.
6. Android蓝牙开发浅谈：http://www.eoeandroid.com/thread-18993-1-1.html.
7. 维基百科“蓝牙”词条：http://zh.wikipedia.org/wiki/蓝牙.
8. 百度百科“蓝牙协议栈”词条：http://baike.baidu.com/view/2015996.htm.
9. 维基百科“蓝牙规范”词条：http://zh.wikipedia.org/zh-cn/蓝牙规范.
10. Android官方文档NFC Basics：http://developer.android.com/guide/topics/nfc/nfc.html.
11. 维基百科“近场通信”词条：http://zh.wikipedia.org/wiki/近场通信.
12. TCP、IP、HTTP、SOCKET的区别和联系：http://www.cnblogs.com/lavenderone/archive/2011/10/14/2212523.html.

第9章 Android WebKit

9.1 Web 2.0/3.0 技术及应用简介

9.1.1 Web 2.0

图 9-1 展示了 Web 技术发展的各个阶段之间的变迁和特点对比。

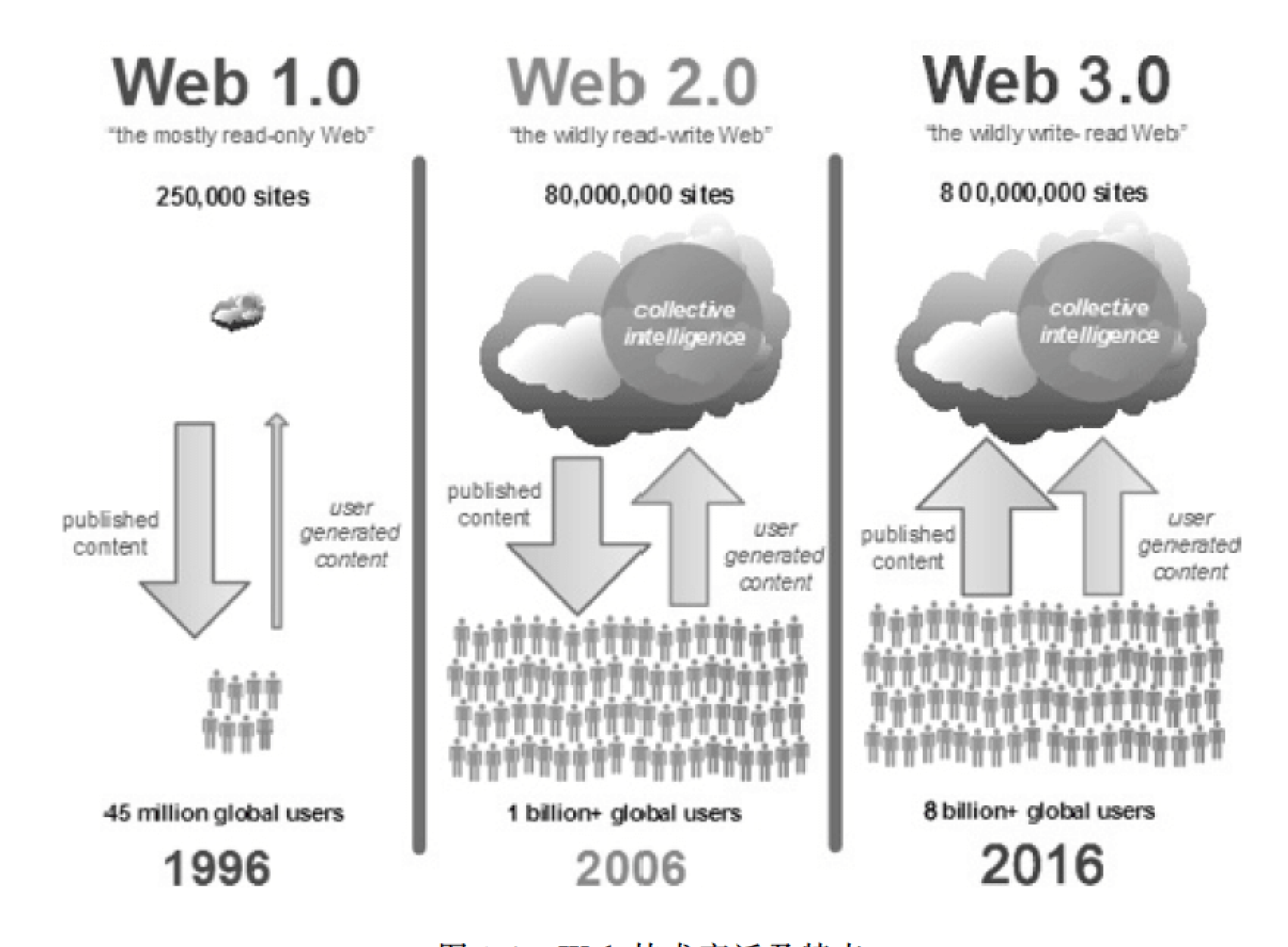


图 9-1　Web 技术变迁及特点

Web 2.0 是以 Flickr、Craigslist、Linkedin、Tribes、Ryze、Friendster、Del. icio. us、43Things. com 等网站为代表，以 BLOG、TAG、SNS、RSS、WIKI 等社会软件的应用为核心，依据六度分隔、xml、ajax 等新理论和技术实现的互联网新一代模式。

1. BLOG

Blog 是 web 和 log 的组合词，它是一种网站或网站的一部分，能够时时更新内容。通常由个人维护，提供时事评论、活动介绍或者其他图片或视频资源。这些条目通常是按照时

间反序排列的。Blog 也可以作为动词,表示维护或增加新内容到 Blog 上。

虽然不是必须的,但是大多数 Blog 是互动的,允许访问者留言、评论。正是这种互动性,使得 Blog 区别于其他静态网站。在这个意义上,Blog 也可以看做社交网络的一种。确实,Blog 拥有者不仅可以创作 Blog 也在同时同他/她的读者或者其他 Blog 拥有者建立了各种社会关系。

许多 Blog 提供了对于某一特定主题的时事评论,其他的功能如在线日记,还有如在线个人或公司品牌广告。一个典型的 Blog 由文字、图片、链接到其他 Blog 的链接、网页以及话题相关媒体组成。读者可以一种互动的形式留言或评论。

2. SNS

SNS,全称为 Social Networking Services,即社会性网络服务,专指旨在帮助人们建立社会性网络的互联网应用服务。也指社会现有已成熟普及的信息载体,如短信 SMS 服务。SNS 的另一种常见解释为:全称 Social Network Site,即“社交网站”或“社交网”。社会性网络(Social Networking)是指个人之间的关系网络,这种基于社会网络关系系统思想的网站就是社会性网络网站(SNS 网站)。SNS 也指 Social Network Software,社会性网络软件,是一种采用分布式技术(通俗地说是采用 P2P 技术)构建的下一代基于个人的网络基础软件。

现在一般所谓的 SNS,则其含义还远不及“熟人的熟人”这个层面。比如根据相同话题进行凝聚(如贴吧)、根据爱好进行凝聚(如 Fexion 网)、根据学习经历进行凝聚(如 Facebook,校内)、根据周末出游的相同地点进行凝聚等,都被纳入“SNS”的范畴。

3. RSS

起源于 RDF Site Summary,又通常认为是 Really Simple Syndication 的缩写。它是一族 Web feed 格式,用于发布频繁更新的内容——包括博客条目,新闻头条,音频和视频——以一种标准化的格式。一个 RSS 文档通常称为“数据资料”或“频道”,包括全文形式或摘要形式的文本,加上元数据,比如发布日期、原创作者等。

RSS 带给发布者很大好处,它可是实现内容的自组织。一个标准的 XML 文件格式即可实现信息的一次发布多次读取(被不同的程序)。同样 RSS 也给读者带来便利,订阅者可以从喜欢的网站上获得实时更新,或者可以把许多不同网站的 feed 集中到一个地方。

RSS feed 可以被一种叫做 RSS 阅读器的软件来读取,阅读器可以是网络软件的,也可以是桌面软件,也可以是移动设备软件。订阅者通过往阅读器中输入 feed 的 URI 或者直接点击浏览器中的 feed 图标,RSS 阅读器检查已订阅的 feed 更新,下载所能找到的更新,然后返回给用户界面。RSS 使得用户不必手动逐个检查各个网站的更新。

4. WIKI

Wiki 是一种网站,用户可以使用一种简化的标记语言或富文本编辑器,通过浏览器添加、修改或删除其内容。Wiki 技术上主要由 Wiki 软件来支持,并且通常通过多个用户协作的方式来完成。如社区网站、企业内部网、知识管理系统,还有笔记备忘。

Wiki适用于许多不同的场景，提供不同功能访问的权限控制。例如，编辑权限可以更改、增加和移除。

9.1.2 Web 3.0

假如说Web 1.0的本质是联合，那么Web 2.0的本质就是互动，它让网民更多地参与信息产品的创造、传播和分享，而这个过程是有价值的。Web 2.0的缺点是没有体现出网民劳动的价值，所以Web 2.0很脆弱，缺乏商业价值。Web 2.0是脆弱的，纯粹的Web 2.0会在商业模式上遭遇重大挑战，需要与具体的产业结合起来才会获得巨大的商业价值和商业成功。Web 3.0是在Web 2.0的基础上发展起来的，能够更好地体现网民的劳动价值，并且能够实现价值均衡分配的一种互联网方式。

Web 3.0的重要特征为：

- 网站内信息可以直接和其他网站信息进行交互，能通过第三方信息平台同时对多家网站信息进行整合使用。
- 用户在互联网上拥有自己的数据，并能在不同的网站上使用。
- 完全基于Web，用浏览器即可实现复杂的系统程序才具有的功能。

Blog，将演变为个人中心，个人中心中的所有内容只有一个域名和一个页面，剩下的所有的服务都由专业服务商提供，用户只需将需要的应用以Widget的方式添加到自己的页面上，就可以享用各种各样完善的服务。但也不会是像Google ig、netvibes这样的集中型个人主页，因为它们没有个性，灵活性也不够。也不会是Sohu这样的Blog平台，因为各种服务都不是一家公司提供的，BSP(Blog Service Provider)可能回归到最原始的个人主页服务，提供一个二级域名和一个静态空间。那么最主要的一个问题：账号由谁提供呢？OpenID肯定会成为Web 3.0的中坚力量，将各个平台有机地连接起来，使你无论走到哪里，都用同一个账号，内容处处关联。而ID服务本身是需要跟信用挂钩的，这是虚拟和现实之间必须建立的桥梁，现有的社区中信用服务都是依靠某种技术手段建立，都很复杂，而且无法跟现实中的人和信用建立起完整有效的关联，不难想到，直接掌握最可靠信用的是银行，所以未来提供OpenID或者互联网身份服务将是银行建立的一种服务，很可能成为银行的某种业务。在这种模式下，互联网服务已经与传统的服务行业一样，提供专业服务，然后收费，互联网的盈利模式也将随之改变。

9.2 WebKit引擎

9.2.1 WebKit简介

WebKit是一个开源的浏览器网页排版引擎，包含WebCore排版引擎和JSCore引擎。WebCore和JSCore引擎来自于KDE项目的KHTML和KJS开源项目。

Google Chrome和Apple Safari正是依靠着WebKit的武装而得以能成为功能强大的现代浏览器。截至2011年10月，WebKit占领了超过33%的浏览器市场。Webkit也被用

于一些试验型浏览器，包括 Amazon Kindle 电子书阅读器，以及 iOS 和 Android 移动操作系统的默认浏览器。WebKit 引擎提供一整套在窗口中显示网页内容的类，同时实现一些浏览器特性，诸如追踪客户点击的链接、管理浏览历史等。

WebKit 最初来源于 Apple 公司的 Konqueror 浏览器的 KHTML 软件库，用以 Safari 浏览器的引擎。现在由一些来自 KDE、Apple Inc.、Nokia、Google、Bitstream、Torch Mobile、Samsung、Igalia 及其他公司和团体的成员共同合作进一步开发。这个项目得到了 Mac OS X、Windows、GNU/Linux 以及其他一些类 UNIX 系统的支持。

WebCore 是一个用于 HTML 和 SVG 的排版、渲染和 DOM 库，由 WebKit 项目开发和维护。WebKit 框架包括 WebCore 和 JavaScriptCore，为基于 C++的 WebCore 渲染引擎和 JavaScript 脚本引擎提供一个 Objective-C 应用编程界面，使得其自身可以很容易地被基于 Cocoa API 的应用程序所采用。JavaScriptCore 是一个框架，它提供了让 WebKit 实现的 JavaScript 引擎，最初 JavaScriptCore 来源于 KDE 的 JavaScript 引擎库和 PCRE 的正则表达式库。从 KJS 分离出来之后，JavaScriptCore 被加入了更多新的特性，性能得到显著提高。

2008 年 6 月 2 日，WebKit 项目宣布重写的 JavaScriptCore 命名为 squirrelfish，是一个字节码解释器。此外 WebKit 还有两个组件 Drosera 和 SunSpider。Drosera 是一个 JavaScript 调试器，SunSpider 是一个基准测试程序套件。

WebKit 通过了 Acid2 和 Acid3 测试。

9.2.2 Android 中的 WebKit 目录和框架

Android 平台的 Web 引擎框架采用了 WebKit 项目中的 WebCore 和 JSCore 部分，上层由 Java 语言封装，并且作为 API 提供给 Android 应用开发者，而底层使用 WebKit 核心库(WebCore 和 JSCore)进行网页排版。

Android 平台的 WebKit 模块分成 Java 层和 WebKit 库两个部分，Java 层负责与 Android 应用程序进行通信，而 WebKit 类库负责实际的网页排版处理。Java 层和 C 层库之间通过 JNI 和 Bridge 相互调用。其模块结构如图 9-2 所示。

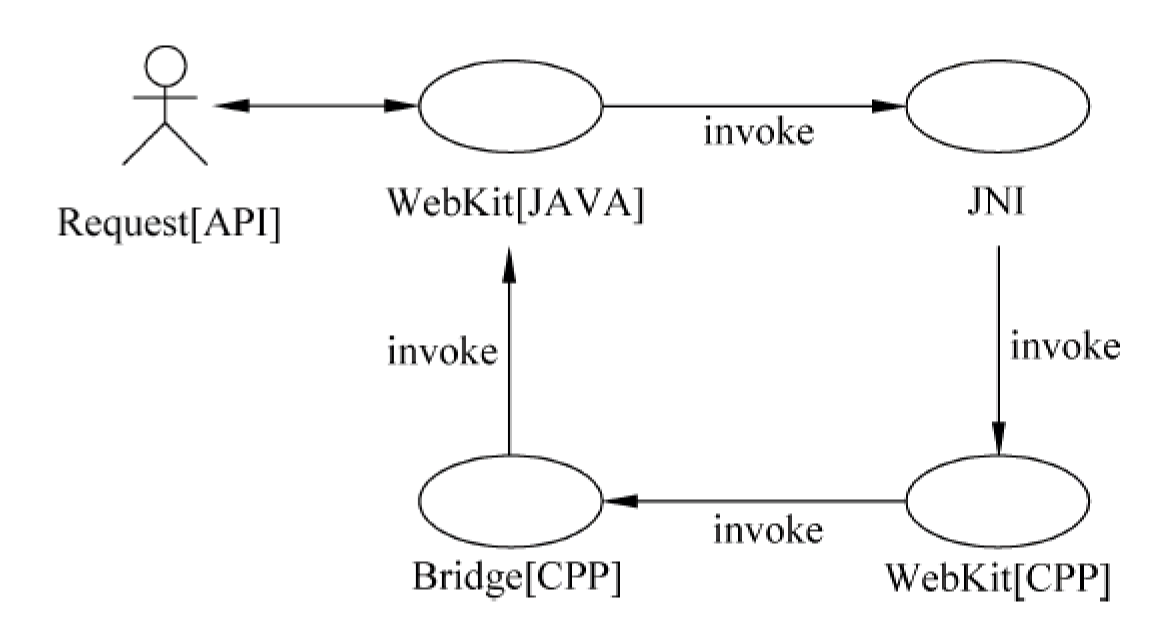


图 9-2 Android Webkit 模块结构图

Android Webkit 的相关开发主要集中在 Java 层，这就有必要了解 Android Webkit 提供的开发 API，相关 API 如表 9-1～表 9-3 所示。

表 9-1 Java 层主要接口及说明

接口	说明
DownloadListener	下载侦听器接口
GeolocationPermissions. Callback	浏览器使用的回调接口,用以报告用户设置的地理位置许可状态
Plugin. PreferencesClickHandler	
PluginStub	用来在 WebView 中实现插件
ValueCallback<T>	回调接口,用来异步地返回值
WebChromeClient. CustomViewCallback	主机应用程序使用的回调接口,通知当前页面,其自定义视图被摒弃了
WebIconDatabase. IconListener	用来接收数据库图标的接口
WebStorage. QuotaUpdater	当一个新的配额生效时,封装一个欲执行的回调函数
WebView. PictureListener	侦听图片的变化

表 9-2 Java 层主要的类及说明

类	说明
CacheManager	Cache 管理对象,负责 Java 层 Cache 对象管理
CacheManager. CacheResult	代表一个从 HTTP cache 取得的资源
ConsoleMessage	WebCore 传来的 JavaScript 控制台信息
CookieManager	根据 RFC2109 规范,管理 Cookies
CookieSyncManager	Cookies 同步管理对象,该对象负责同步 RAM 和 Flash 间的 Cookies 数据。实际的物理数据操作在基类 WebSyncManager 中完成
DateSorter	对日期排序(分为:今天、昨天、一周前、一月前、早于一月前)
GeolocationPermissions	取得、设置地理位置许可
HttpAuthHandler	Http 认证处理对象,该对象会作为参数传递给 BrowserCallback. displayHttpAuthDialog 方法,与用户交互
JsPromptResult	Js 结果提示对象,用于向用户提示 JavaScript 运行结果
JsResult	Js 结果对象,用于用户交互
MimeTypeMap	将 MIME 类型映射到相应的文件扩展名,或反之
SslErrorHandler	用于处理 SSL 错误消息
URLUtil	URL 处理功能函数,用于编码、解码 URL 字符串,以及提供附加的 URL 类型分析功能
WebBackForwardList	该对象包含 WebView 对象中显示的历史数据
WebChromeClient	Chrome 客户基类,Chrome 客户对象在浏览器文档标题、进度条、图标改变时会得到通知
WebHistoryItem	该对象用于保存一条网页历史数据
WebIconDatabase	图标数据库管理对象,所有的 WebView 均请求相同的图标数据库对象
WebSettings	WebView 的管理设置数据,该对象数据是通过 JNI 接口从底层获取

表 9-3　Java 层主要枚举及说明

枚　　举	说　　明
ConsoleMessage. MessageLevel	含 DEBUG、ERROR、LOG、TIP、WARNING
WebSettings. LayoutAlgorithm	控制 htmld 布局 含 NARROW_COLUMNS、NORMAL、SINGLE_COLUMN
WebSettings. PluginState	影响如何对待页面上的插件 含 OFF、ON、ON_DEMAND
WebSettings. RenderPriority	含 HIGH、LOW、NORMAL
WebSettings. TextSize	含 LARGER、LARGEST、NORMAL、SMALLER、SMALLEST
WebSettings. ZoomDensity	指定 WebView 的最佳分辨率 含 CLOSE、FAR、MEDIUM

9.3　Android 上的 WebKit 开发

9.3.1　基本开发

下面介绍 Java 层常用的重要类：

- WebView——专门用来浏览网页。
- WebSettings——WebSettings 来设置 WebView 的一些属性、状态等。
- WebViewClient——WebViewClient 帮助 WebView 处理各种通知、请求事件。
- WebChromeClient——WebChromeClient 辅助 WebView 处理 JavaScript 的对话框、网站图标、网站标题、加载进度等。

WebViewClient 和 WebChromeClient 可以被看作是辅助 WebView 管理网页中各种通知、请求等事件以及 JavaScript 事件的两个类。

1. 使用 WebView 初始化 WebKit

WebView 常用的重要方法：

- void addJavascriptInterface(Object obj, String interfaceName)；该方法将一个 Java 对象绑定到一个 JavaScript 对象中，JavaScript 对象名就是 interfaceName，作用域是 Global。从而可由网页中的 JavaScript 来操作 Java 对象。
- WebSettings getSettings()；该方法返回 WebView 对象相对应的 WebSettings 对象。
- void loadData(String data, String mimeType, String encoding)；该方法往往用来加载 html 数据，它不能通过网络来加载内容。
- void loadUrl(String url)；该方法通过字符串参数给出欲加载的页面的 URL。
- void setWebChromeClient(WebChromeClient client)；该方法设置 Chrome 客户对象。
- void setWebViewClient(WebViewClient client)；该方法设置 Web 视图客户对象。

例 9-1　WebView 简单示例。

(1) 在 Eclipse 中新建 Android Project，如图 9-3 所示。

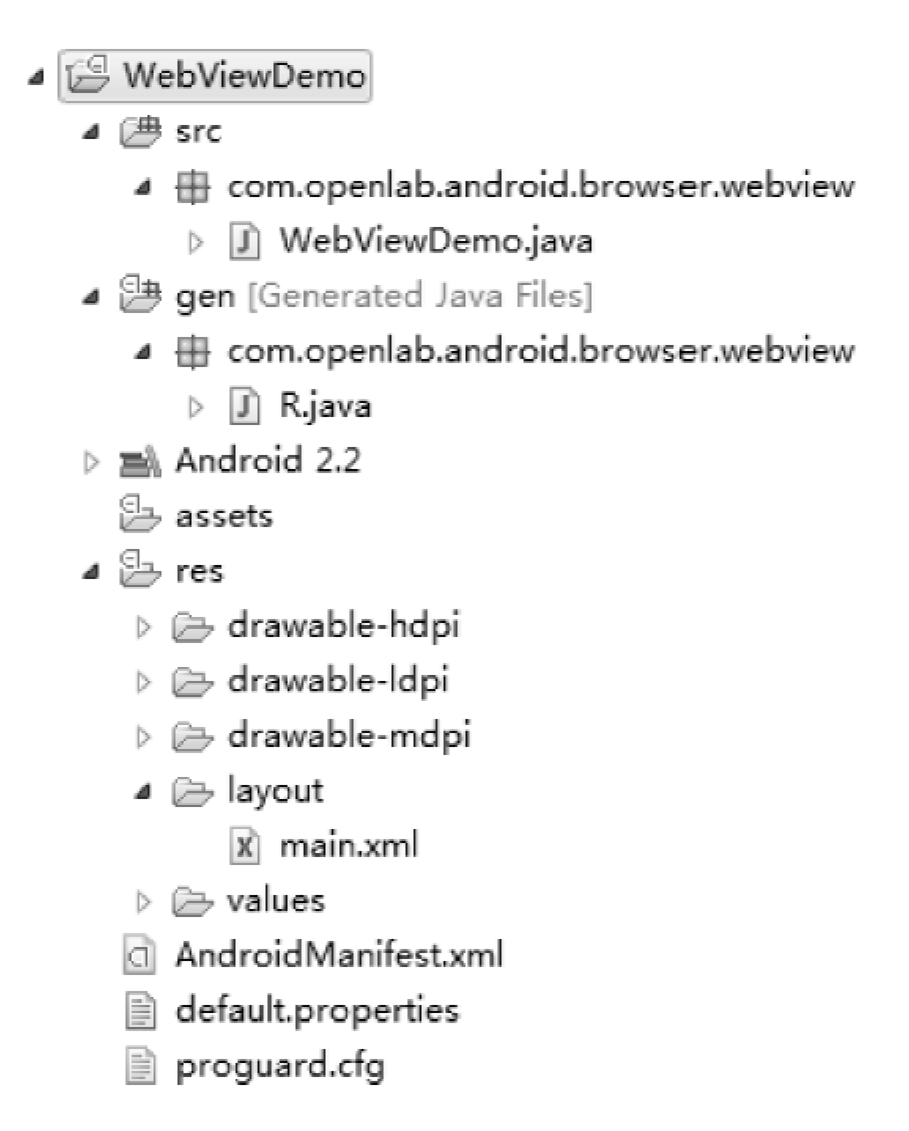


图 9-3 WebViewDemo 项目结构

(2) 打开 res/layout/main.xml 文件,拖入 WebView 组件,如图 9-4 所示。

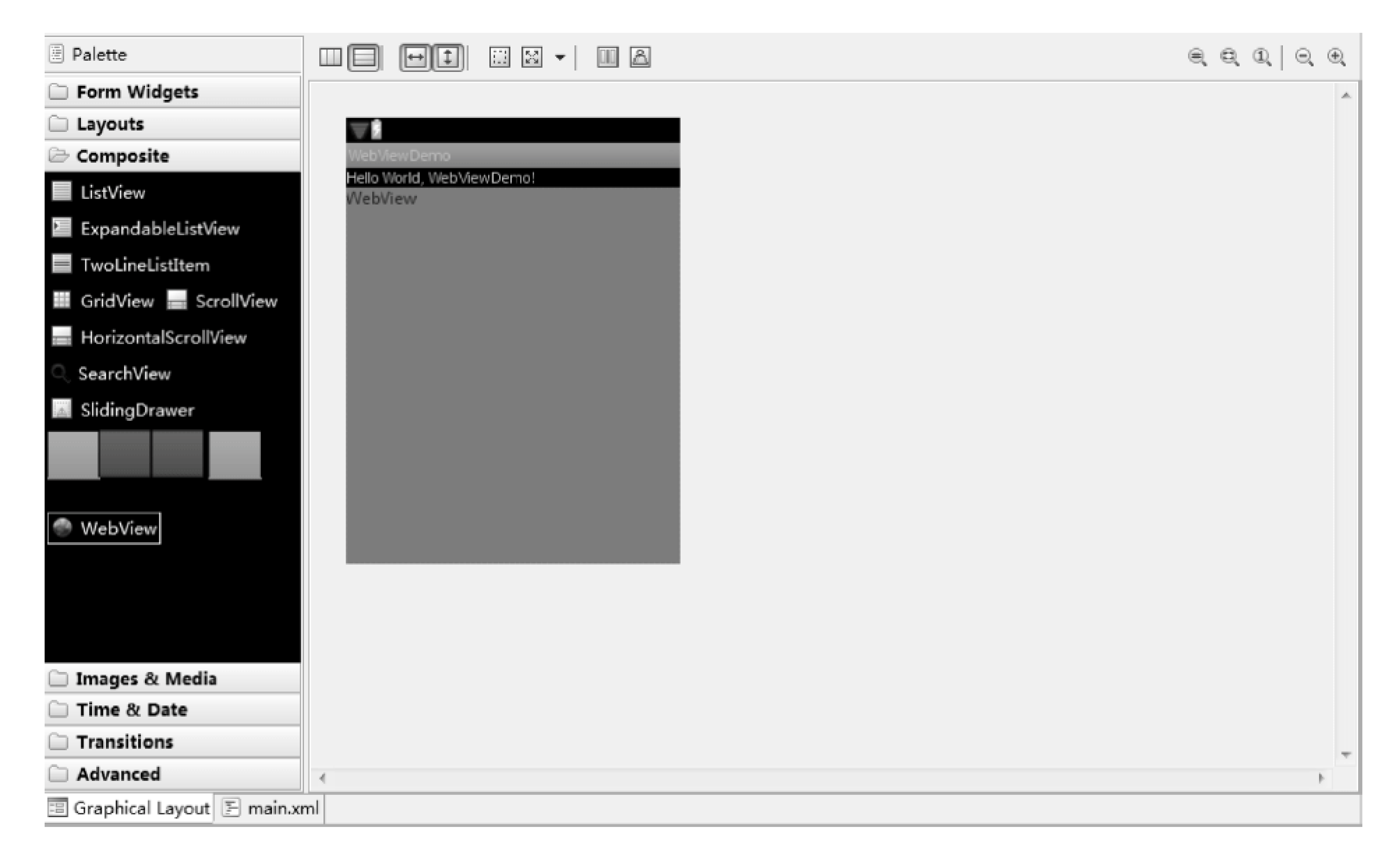


图 9-4 Eclipse Android 图形设计界面

这时 main.xml 文件的内容为:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent" android:layout_height="fill_parent"
android:orientation="vertical">
```

```
<TextView android:layout_height = "wrap_content"
    android:text = "@string/hello" android:layout_width = "fill_parent" />
<WebView android:id = "@+ id/webView1" android:layout_width = "match_parent"
    android:layout_height = "match_parent"></WebView>
</LinearLayout>
```

（3）打开 WebViewDemo.java 文件，添加 WebView 组件的使用代码，完整代码如下：

```
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class WebViewDemo extends Activity {
/ ** Called when the activity is first created. * /
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    WebView view = (WebView) findViewById(R.id.webView1);
    view.loadurl("http://m.renren.com/");
}
}
```

（4）打开 AndroidManifest.xml 文件，添加权限，如图 9-5 所示。

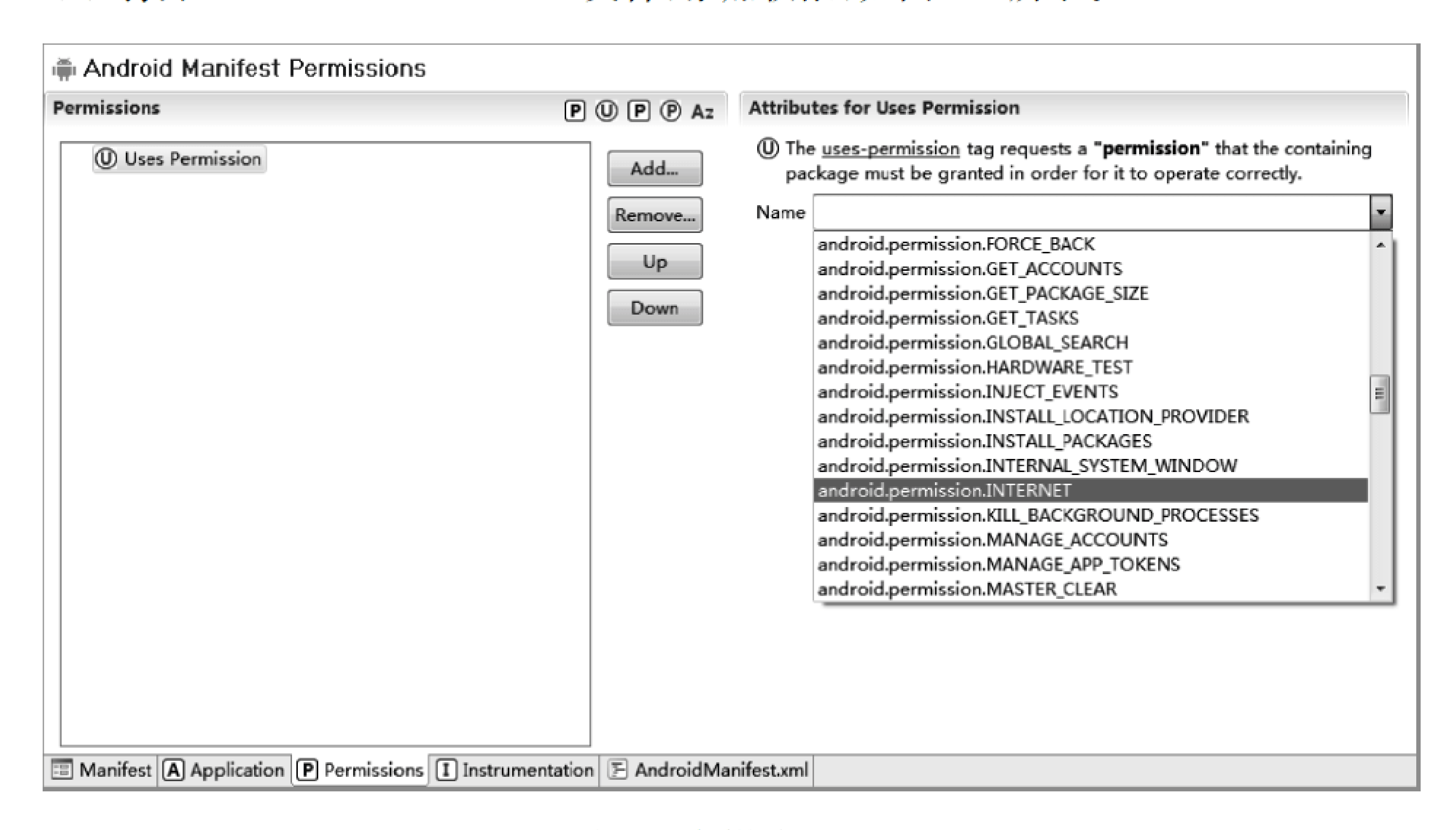


图 9-5 权限设置界面

这时 AndroidManifest.xml 文件的内容为：

```
<?xml version = "1.0" encoding = "utf - 8"?>
<manifest xmlns:android = "http://schemas.android.com/apk/res/android"
```

```
package="com.openlab.android.browser.webview" android:versionCode="1"
android:versionName="1.0">
<uses-permission android:name="android.permission.INTERNET"></uses-permission>

<application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".WebViewDemo" android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

</application>
</manifest>
```

(5) 右击项目,在弹出的快捷菜单中选择 Run As Android Application 命令,把项目安装进模拟器并运行,如图 9-6 所示。

2. 使用 WebSettings 设置 WebView 的属性和状态

WebSettings 的常用方法如下:

- void setAllowFileAccess(boolean allow);//设置是否允许访问文件数据
- void setBuiltInZoomControls(boolean enabled);//设置是否显示放大缩小 controler
- synchronized void setJavaScriptEnabled(boolean flag);//设置是否支持 JavaScript
- void setSaveFormData(boolean save);//设置是否保存表单数据
- void setSavePassword(boolean save);//设置是否保存密码
- void setSupportZoom(boolean support);//设置是否支持缩放



图 9-6 运行结果界面

例 9-2 使用 WebSettings 设置 WebView 的属性和状态。

例 1 完成了一个最简单的 WebView 使用实例。但是如果访问的页面中有 Javascript,则 WebView 必须设置支持 Javascript。这就会使用到 WebSettings 的方法。

```java
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class WebViewDemo extends Activity {
```

```
/ ** Called when the activity is first created. * /
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    WebView view = (WebView) findViewById(R.id.webView1);

    WebSettings webSettings = view.getSettings();
    webSettings.setJavaScriptEnabled(true);
    view.loadurl("http://m.renren.com/");
}
}
```

3. 使用 WebViewClient 处理 WebView 的通知和事件

WebViewClient 的常用方法如下：

- void onPageFinished(WebView view, String url);//页面加载完毕时执行
- void onPageStarted(WebView view, String url, Bitmap favicon);//页面加载开始时执行
- void onReceivedError(WebView view, int errorCode, String description, String fallingUrl);//向主机应用程序报告错误
- boolean shouldOverrideUrlLoading(WebView view, String url);//控制新的连接是否在当前 WebView 中打开

//参数 view 表示初始化该调用的 WebView
//返回真,若主机应用程序欲离开当前 WebView,自行处理 url
//否则返回假

例 9-3 使用 WebViewClient 处理 WebView 的通知和事件。

例 2 在例 1 的基础上添加了支持 JavaScript 的功能,但是如果页面中链接,现在点击链接后会启动 Android 的系统 browser 来响应该链接；如果希望继续在当前 browser 中响应,则必须覆盖 WebView 的 WebViewClient 对象。

```
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;

public class WebViewDemo extends Activity
/ ** Called when the activity is first created. * /
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    WebView view = (WebView) findViewById(R.id.webView1);
```

```
    WebSettings webSettings = view.getSettings();
    webSettings.setJavaScriptEnabled(true);

    view.setWebViewClient(new WebViewClient() {
        public boolean shouldOverrideUrlLoading(WebView view, String url) {
            view.loadUrl(url);
            return true;
        }
    });

    view.loadurl("http://m.renren.com/");
}
}
```

4. 使用 WebChromeClient 处理 WebView 的加载进度

WebChromeClient 常用方法：

- boolean onJsAlert(WebView view, String url, String message, JSResult result);
  //告诉 Chrome 客户对象显示一个 javascript 的警告对话框
  //参数 result: A JsResult to confirm that the user hit enter
- boolean onJsConfirm(WebView view, String url, String message, JSResult result);
  //告诉 Chrome 客户对象显示一个确认对话框
  //参数 result: A JsResult used to send the user's response to javascript
- boolean onJsPrompt(WebView view, String url, String message, String defaultValue, JSResult result);
  //告诉 Chrome 客户对象显示一个提示对话框
  //参数 default: The default value displayed in the prompt dialog
  //参数 result: A JsPromptResult used to send the user's reponse to javascript
- void onProgressChanged(WebView view, int newProgress);
  //告诉主机应用程序,加载页面的当前进度
  //参数 newProgress: Current page loading progress, represented by an integer (between 0 and 100)

例 9-4　使用 WebChromeClient 处理 WebView 的加载进度。

一些网页载入有些慢,若希望有个进度条显示,需要使用另外一个线程来加载网页并且使用 WebChromeClient 对象和 handler 来处理消息,效果如图 9-7 所示。



图 9-7　运行结果界面

```java
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.app.ProgressDialog;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.webkit.WebChromeClient;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;

public class WebViewDemo extends Activity {
WebView view;
ProgressDialog dialog;

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    view = (WebView) findViewById(R.id.webView1);

    WebSettings webSettings = view.getSettings();
    webSettings.setJavaScriptEnabled(true);

    view.setWebViewClient(new WebViewClient() {
        public boolean shouldOverrideUrlLoading(WebView view, String url) {
            view.loadUrl(url);
            return true;
        }
    });

    view.setWebChromeClient(new WebChromeClient() {
        public void onProgressChanged(WebView view, int progress) {
            if (progress == 100) {
                viewHandler.sendEmptyMessage(1);
            }
            super.onProgressChanged(view, progress);
        }
    });

    dialog = new ProgressDialog(WebViewDemo.this);
    dialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
    dialog.setMessage("数据载入中,请稍候!");

    loadurl(view, "http://m.renren.com/");
}

public Handler viewHandler = new Handler() {
    public void handleMessage(Message msg) {
        if (!Thread.currentThread().isInterrupted()) {
            switch (msg.what) {
```

```
            case 0:
                dialog.show();
                break;
            case 1:
                dialog.hide();
                break;
            }
            super.handleMessage(msg);
        }
    }
};

public void loadurl(final WebView view, final String url) {
    new Thread() {
        public void run() {
            viewHandler.sendEmptyMessage(0);
            view.loadUrl(url);
        }
    }.start();
}
}
```

到目前为止，已经掌握了 WebView 主要的使用方法。下面进入高级开发，在改进前面例子实现的 WebView 的同时，巩固和加强 Android Browser 的编程方法和技巧。

9.3.2　高级开发

1. 处理退出事件

例 9-5　处理退出事件。

在前面的例子中已完成的 WebView 中浏览网页点击系统 Back 键时，整个 Browser 会调用 finish() 而结束自身，如果希望浏览的网页回退而不是退出浏览器，需要在当前 Activity 中处理 goBack 事件。

```
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.app.ProgressDialog;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.KeyEvent;
import android.webkit.WebChromeClient;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;

public class WebViewDemo extends Activity {
WebView view;
ProgressDialog dialog;
```

```

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    view = (WebView) findViewById(R.id.webView1);

    WebSettings webSettings = view.getSettings();
    webSettings.setJavaScriptEnabled(true);

    view.setWebViewClient(new WebViewClient() {
        public boolean shouldOverrideUrlLoading(WebView view, String url) {
            view.loadUrl(url);
            return true;
        }
    });

    view.setWebChromeClient(new WebChromeClient() {
        public void onProgressChanged(WebView view, int progress) {
            if (progress == 100) {
                viewHandler.sendEmptyMessage(1);
            }
            super.onProgressChanged(view, progress);
        }
    });

    dialog = new ProgressDialog(WebViewDemo.this);
    dialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
    dialog.setMessage("数据载入中,请稍候!");

    loadurl(view, "http://m.renren.com/");
}

public boolean onKeyDown(int keyCode, KeyEvent event) {
    if ((keyCode == KeyEvent.KEYCODE_BACK) && view.canGoBack()) {
        view.goBack();
        return true;
    }
    return super.onKeyDown(keyCode, event);
}

public Handler viewHandler = new Handler() {
    public void handleMessage(Message msg) {
        if (!Thread.currentThread().isInterrupted()) {
            switch (msg.what) {
            case 0:
                dialog.show();
                break;
            case 1:
                dialog.hide();
                break;
            }
```

```
                super.handleMessage(msg);
            }
        }
};

public void loadurl(final WebView view, final String url) {
    new Thread() {
        public void run() {
            viewHandler.sendEmptyMessage(0);
            view.loadUrl(url);
        }
    }.start();
}
}
```

虽然经过这样的改进，但是如果回退到第一个网页后，再回退还是会结束浏览器，通常这不是我们所希望的，需要在当前 Activity 中处理，实现效果如图 9-8 所示。



图 9-8 运行结果界面

```
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.KeyEvent;
```

```java
import android.webkit.WebChromeClient;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;

public class WebViewDemo extends Activity
{
WebView view;
ProgressDialog dialog;

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    view = (WebView) findViewById(R.id.webView1);

    WebSettings webSettings = view.getSettings();
    webSettings.setJavaScriptEnabled(true);

    view.setWebViewClient(new WebViewClient() {
        public boolean shouldOverrideUrlLoading(WebView view, String url)
        {
            view.loadUrl(url);
            return true;
        }
    });

    view.setWebChromeClient(new WebChromeClient() {
        public void onProgressChanged(WebView view, int progress)
        {
            if (progress == 100)
            {
                viewHandler.sendEmptyMessage(1);
            }
            super.onProgressChanged(view, progress);
        }
    });

    dialog = new ProgressDialog(WebViewDemo.this);
    dialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
    dialog.setMessage("数据载入中,请稍候!");

    loadurl(view, "http://m.renren.com/");
}

public boolean onKeyDown(int keyCode, KeyEvent event)
{
    if (keyCode == KeyEvent.KEYCODE_BACK)
```

```
    {
        if (view.canGoBack())
        {
            view.goBack();
        }
        else
        {
            ConfirmExit();
        }
        return true;
    }
    return super.onKeyDown(keyCode, event);
}

public void ConfirmExit()
{
    AlertDialog.Builder ad = new AlertDialog.Builder(WebViewDemo.this);
    ad.setTitle("退出");
    ad.setMessage("是否退出软件?");
    ad.setPositiveButton("是", new DialogInterface.OnClickListener() {

        @Override
        public void onClick(DialogInterface dialog, int which)
        {
            WebViewDemo.this.finish();
        }
    });
    ad.setNegativeButton("否", new DialogInterface.OnClickListener() {

        @Override
        public void onClick(DialogInterface dialog, int which)
        {
        }
    });

    ad.show();
}

public Handler viewHandler = new Handler() {
    public void handleMessage(Message msg)
    {
        if (!Thread.currentThread().isInterrupted())
        {
            switch (msg.what)
            {
                case 0:
                    dialog.show();
                    break;
                case 1:
                    dialog.hide();
```

```
                        break;
                }
                super.handleMessage(msg);
            }
        }
};

public void loadurl(final WebView view, final String url)
{
    new Thread() {
        public void run()
        {
            viewHandler.sendEmptyMessage(0);
            view.loadUrl(url);
        }
    }.start();
}
}
```

2．浏览器雏形

例 9-6 浏览器雏形。

现在的 WebView 最不像浏览器的地方是没有地址栏，如果希望有地址栏，需要自己添加 EditText 和 Button 组件，调用 loadUrl 方法，操作过程如图 9-9 和图 9-10 所示。

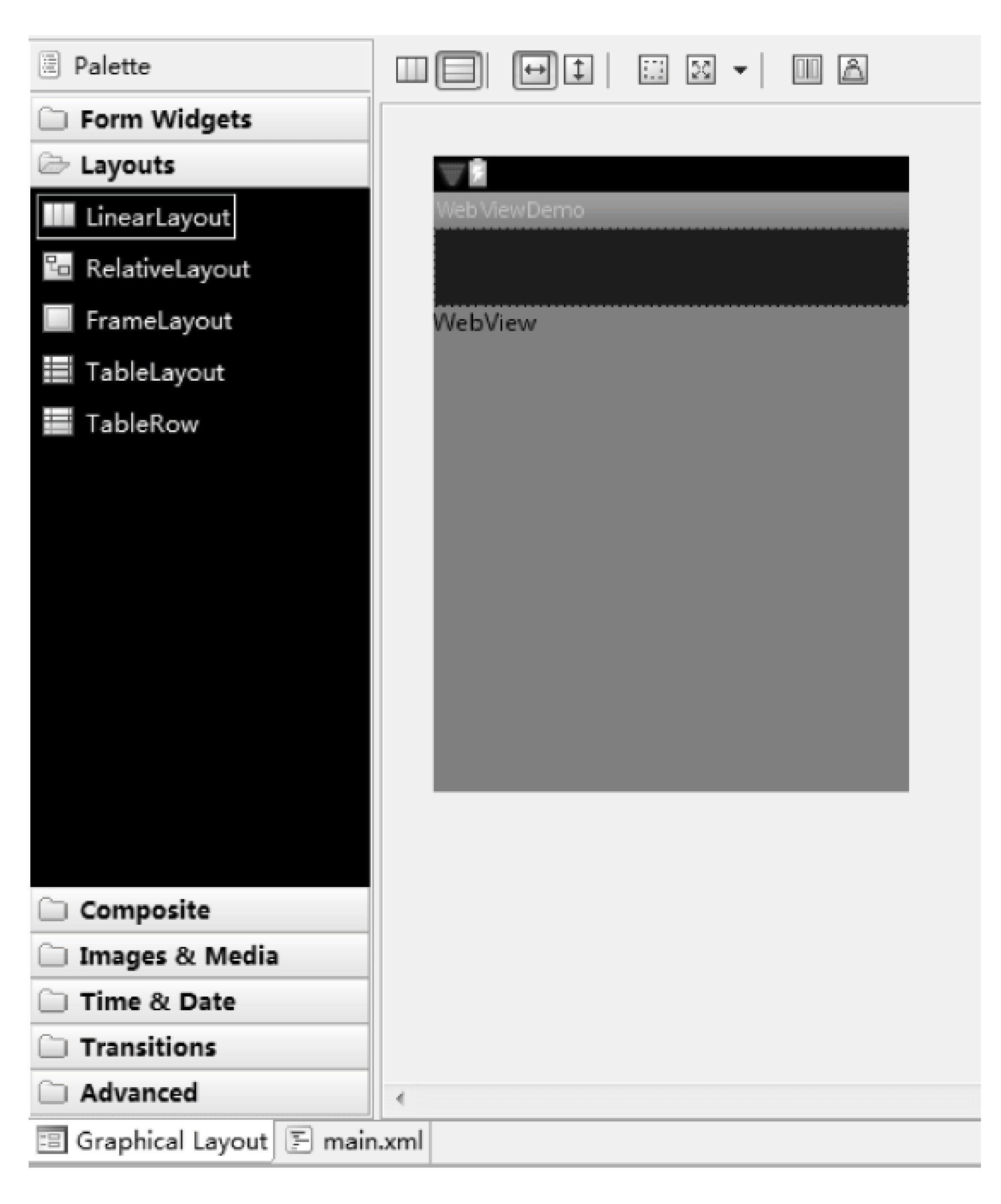


图 9-9 添加组件界面



图 9-10　编辑组件界面

```
package com.openlab.android.browser.webview;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnClickListener;
import android.webkit.WebChromeClient;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class WebViewDemo extends Activity
{
WebView view;
ProgressDialog dialog;
```

```

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    view = (WebView) findViewById(R.id.webView1);

    WebSettings webSettings = view.getSettings();
    webSettings.setJavaScriptEnabled(true);

    view.setWebViewClient(new WebViewClient() {
        public boolean shouldOverrideUrlLoading(WebView view, String url)
        {
            view.loadUrl(url);
            return true;
        }
    });

    view.setWebChromeClient(new WebChromeClient() {
        public void onProgressChanged(WebView view, int progress)
        {
            if (progress == 100)
            {
                viewHandler.sendEmptyMessage(1);
            }
            super.onProgressChanged(view, progress);
        }
    });

    dialog = new ProgressDialog(WebViewDemo.this);
    dialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
    dialog.setMessage("数据载入中,请稍候!");

    final EditText urlText = (EditText) findViewById(R.id.editText1);
    urlText.setText("m.renren.com");
    Button urlButton = (Button) findViewById(R.id.button1);
    urlButton.setText("进入");
    urlButton.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v)
        {
            String url = urlText.getText().toString();
            if (url == null || "".equals(url))
            {
                Toast.makeText(WebViewDemo.this, "请输入 URL",
                        Toast.LENGTH_SHORT).show();
            }
            else
            {
                if (!url.startsWith("http:"))
                {
```

```
                    url = "http://" + url;
                }
                loadurl(view, url);
            }
        }
    });

    loadurl(view, "http://m.renren.com/");
}

public boolean onKeyDown(int keyCode, KeyEvent event)
{
    if (keyCode == KeyEvent.KEYCODE_BACK)
    {
        if (view.canGoBack())
        {
            view.goBack();
        }
        else
        {
            ConfirmExit();
        }
        return true;
    }
    return super.onKeyDown(keyCode, event);
}

public void ConfirmExit()
{
    AlertDialog.Builder ad = new AlertDialog.Builder(WebViewDemo.this);
    ad.setTitle("退出");
    ad.setMessage("是否退出软件?");
    ad.setPositiveButton("是", new DialogInterface.OnClickListener() {

        @Override
        public void onClick(DialogInterface dialog, int which)
        {
            WebViewDemo.this.finish();
        }
    });
    ad.setNegativeButton("否", new DialogInterface.OnClickListener() {

        @Override
        public void onClick(DialogInterface dialog, int which)
        {
        }
    });

    ad.show();
}

public Handler viewHandler = new Handler() {
    public void handleMessage(Message msg)
```

```
    {
        if (!Thread.currentThread().isInterrupted())
        {
            switch (msg.what)
            {
                case 0:
                    dialog.show();
                    break;
                case 1:
                    dialog.hide();
                    break;
            }
            super.handleMessage(msg);
        }
    }
};

public void loadurl(final WebView view, final String url)
{
    new Thread() {
        public void run()
        {
            viewHandler.sendEmptyMessage(0);
            view.loadUrl(url);
        }
    }.start();
}
}
```

3. 支持 HTTPS 协议

WebView 默认是不处理 HTTPS 请求的,需要进行如下设置:

```
view.setWebViewClient(new WebViewClient() {
        public boolean shouldOverrideUrlLoading(WebView view, String url)
        {
            view.loadUrl(url);
            return true;
        }

        @Override
        public void onReceivedSslError(WebView view,
                SslErrorHandler handler, SslError error)
        {
            handler.proceed();
            // handler.cancel();
            // handler.handleMessage(null);
        }

    });
```

onReceivedSslError 为 WebView 处理 ssl 证书设置，其中：

- handler. proceed()；//表示等待证书响应。
- handler. cancel()；//表示挂起连接，为默认方式。
- handler. handleMessage(null)；//可做其他处理。

4. 支持 JavaScript 扩展

1）Javascript 中调用 java 对象及方法

设置 webView 的 addJavascriptInterface 方法，该方法有两个参数：第一个参数为被绑定到 js 中的类实例；第二个参数为在 js 中暴露的类别名，在 js 中引用 Java 对象就是用这个名字。

```
ClassBeBindedToJS classBeBindedToJS = new ClassBeBindedToJS();
webView.addJavascriptInterface(classBeBindedToJS, "classNameBeExposedInJs");
```

实现绑定到 js 的类：

```
private class ClassBeBindedToJS
{
public String javaMethod()
{
    return "use java method";
}
};
```

其中，JavaMethod 方法将在页面前端 js 中调用，用于返回一段内容。

在前端调用 Java 对象：

```
<html>
        <body>
        <div id = "displayDiv"> Test page.</div>
                < input type = " button" value = " use java object" onclick = " document.getElementById('displayDiv').innerHTML = classNameBeExposedInJs.javaMethod()" />
        </body>
</html>
```

单击 button 按钮，改变 div 内容为 Java 对象方法中的内容。

2）Java 调用 Javascript 方法

用 webView 的 loadUrl 方法，加载伪 URL 实现。

```
view.setWebViewClient(new WebViewClient() {
    @Override
    public void onPageFinished(WebView webView, String url)
    {
        webView.loadUrl("javascript:hello()");
    }
});
```

3）用途举例：实现网页自动登录

```
view.getSettings().setJavaScriptEnabled(true);
view.loadUrl("http://m.renren.com");
view.requestFocus();

view.setWebViewClient(new WebViewClient() {
    @Override
    public void onPageFinished(WebView webView, String url)
    {
        webView.loadUrl("javascript:document.getElementsByName('email')[0].value = 'userName'");
        webView.loadUrl("javascript:document.getElementsByName('password')[0].value = 'userPassword'");
        webView.loadUrl("javascript:document.getElementsByName('login')[0].click()");
    }
});
```

例 9-7　添加 JavaScript 扩展。

- 这里使用 apps-for-android 中的实例。

打开 SVN，checkout 地址如下：

http://apps-for-android.googlecode.com/svn/trunk/Samples/WebViewDemo

项目结构如图 9-11 所示。

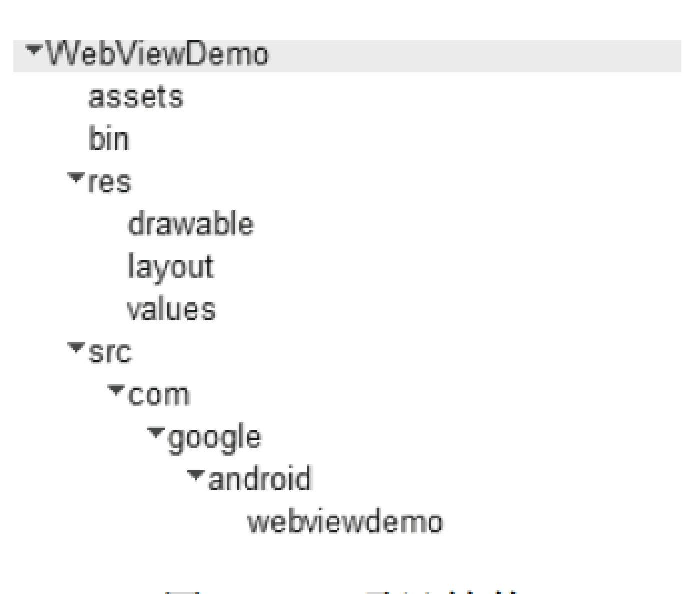


图 9-11　项目结构

- WebViewDemo.java 内容为：

```
package com.google.android.webviewdemo;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.webkit.JsResult;
import android.webkit.WebChromeClient;
import android.webkit.WebSettings;
import android.webkit.WebView;
/**
 * Demonstrates how to embed a WebView in your activity. Also demonstrates how
 * to have javascript in the WebView call into the activity, and how the activity
 * can invoke javascript.
 * <p>
```

```
 * In this example, clicking on the android in the WebView will result in a call into
 * the activities code in {@link DemoJavaScriptInterface#clickOnAndroid()}. This code
 * will turn around and invoke javascript using the {@link WebView#loadUrl(String)}
 * method.
 * <p>
 * Obviously all of this could have been accomplished without calling into the activity
 * and then back into javascript, but this code is intended to show how to set up the
 * code paths for this sort of communication.
 *
 */
public class WebViewDemo extends Activity {
    private static final String LOG_TAG = "WebViewDemo";
    private WebView mWebView;
    private Handler mHandler = new Handler();
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        mWebView = (WebView) findViewById(R.id.webview);
        WebSettings webSettings = mWebView.getSettings();
        webSettings.setSavePassword(false);
        webSettings.setSaveFormData(false);
        webSettings.setJavaScriptEnabled(true);
        webSettings.setSupportZoom(false);
        mWebView.setWebChromeClient(new MyWebChromeClient());
        mWebView.addJavascriptInterface(new DemoJavaScriptInterface(), "demo");
        mWebView.loadUrl("file:///android_asset/demo.html");
    }
    final class DemoJavaScriptInterface {
        DemoJavaScriptInterface() {
        }
        /**
         * This is not called on the UI thread. Post a runnable to invoke
         * loadUrl on the UI thread.
         */
        public void clickOnAndroid() {
            mHandler.post(new Runnable() {
                public void run() {
                    mWebView.loadUrl("javascript:wave()");
                }
            });
        }
    }
    /**
     * Provides a hook for calling "alert" from javascript. Useful for
     * debugging your javascript.
     K*/
    final class MyWebChromeClient extends WebChromeClient {
        @Override
        public boolean onJsAlert(WebView view, String url, String message, JsResult result) {
            Log.d(LOG_TAG, message);
            result.confirm();
            return true;
```

```
            }
        }
}
```

- Demo. html 内容为：

```
<html>
        <script language = "javascript">
            /* This function is invoked by the activity */
                function wave() {
                    alert("1");
                        document.getElementById("droid").src = "android_waving.png";
                        alert("2");
                }
         </script>
         <body>
            <!-- Calls into the javascript interface for the activity -->
            <a onClick = "window.demo.clickOnAndroid()"><div style = "width:80px;
                    margin:0px auto;
                    padding:10px;
                    text-align:center;
                    border:2px solid #202020;" >
                         <img id = "droid" src = "android_normal.png"/><br>
                         Click me!
                </div></a>
        </body>
</html>
```

参考文献

1. WebKit-WebKit For Android：http://www.jjos.org/android/2010/05/10/.
2. Apps-for-android：http://code.google.com/p/apps-for-android/source/browse/# git%2FSamples%2FWebViewDemo.

第10章 NDK入门

10.1 NDK 简介

NDK 的英文全称为 Native Development Kit，即原生开发工具包。它是一种基于原生程序接口的软件开发工具，通过此开发工具所开发的应用程序能够直接在设备上以本地语言运行，而不需要借助于虚拟机。通常类似于 Java 语言的基于虚拟机运行的语言的程序会配套有原生开发工具包。由于基于虚拟机的语言在运行上比基于 C 语言或 C++语言的效率要低，因此通过 NDK 编译的原生程序在一些情况下能够维持运行的高效率，NDK 可以在硬件支持的情况下兼容任何 C 语言的库，因此在功能上能够弥补 SDK 开发的不足。虽然 NDK 开发的程序运行效率更高，但是使用 SDK 开发应用程序在开发效率上会有一定的优势，在 NDK 上开发程序的难度也要比在 SDK 上开发要高。

由于 Android 平台的应用程序正是基于其 Dalvik 虚拟机，因此 Android 提供了 NDK 开发组件。Android NDK 是一套帮助开发人员将本地代码嵌入到 Android 应用程序中去的工具包。Android 应用程序运行在 Dalvik 虚拟机中，使用的是 Java 代码，NDK 允许开发人员将其所开发的应用程序的一部分使用 C/C++本地代码来实现。

10.1.1 Android NDK 组成

Android NDK 是一系列工具的集合，这些工具能够帮助发者快速开发 C/C++动态库，并能自动将 so 和 Java 应用一起打包成 apk。这些工具对开发者的帮助是巨大的。

NDK 集成了交叉编译器，并提供了相应的 mk 文件隔离 CPU、平台、ABI 等差异，开发人员只需要简单修改 mk 文件(指出“哪些文件需要编译”、“编译特性要求”等)，就可以创建出 so。so 文件是一种 ELF(Executable and Linkable Format，可执行连接格式)格式文件，可以将其理解为共享库(或动态库)，类似于 Windows 系统下的 dll 文件。利用 so 可以节约资源，加快速度，使代码升级简化。由于 NDK 可以自动地将 so 和 Java 应用一起打包，极大地减轻了开发人员的打包工作。

10.1.2 NDK API 的性质

NDK 提供了一份稳定但功能有限的 API 头文件声明。Google 明确声明该 API 是稳定的，在后续所有版本中都稳定支持当前发的 API。从现有版本的 NDK 中可以看出，这些 API 支持的功能非常有限，仅仅包含有 C 标准库(libc)、标准数学库(libm)、压缩库(libz)、

Log库(liblog)等。

10.1.3 NDK的作用

使用NDK,开发人员可以将要求高性能的应用逻辑使用C开发,从而提高应用程序的执行效率。使用NDK,可以将需要保密的应用逻辑使用C开发。毕竟Java包都是可以反编译的。另外,NDK促使专业so组件商的出现,可以加速Android平台的发展。

10.1.4 使用NDK的注意事项

即使从某种意义上说使用本地代码也可以编写出功能强大的应用程序,但是使用NDK来开发运行在Android设备上的通用原生代码是一种不可取的方式。通常来说,开发人员还是应该使用Java来开发Android应用程序,这样可以更好地处理Android系统事件(例如如何避免“Application Not Responding”发生)以及更好地控制Android各组件的生命周期。

总体来说,NDK大多数时候并不能够使应用程序受益,因此开发应用程序时最好能够仔细地度量NDK开发的利与弊,由于NDK开发更趋向于下层,因此使用NDK开发出来的代码通常并不能提高其自动化水平,相反会大大增加代码的复杂度。除非在出于绝对必要的情况下,不推荐使用本地代码编程程序,即使开发人员在使用C/C++编程方面更加熟练。

另外,使用NDK开发时还需要注意如下两个问题:

(1) 不能直接地通过本地代码指针去访问VM的对象内容。

(2) 如果本地代码意图通过JNI方法调用去管理VM对象,就需要显式地引用管理。

10.2 Windows下NDK开发环境的搭建

10.2.1 开发环境组成

要在Windows下搭建Android NDK的开发环境,需要安装如下一些应用程序或者组件:

(1) Eclipse集成开发环境——由于Android应用程序通常在Eclipse下进行开发,因此为了使NDK开发环境与SDK开发环境并存,使用Eclipse是最佳的一种方法,它能够使得包含本地代码的程序在编译和调试时更加方便。

(2) Eclipse下用于C/C++开发的CDT插件——CDT插件是用于支持在Eclipse下开发和调试C/C++程序的插件,借助于CDT,可以配置和使用GCC、MinGW、Cygwin等编译器来编译和链接代码,使用CDT来编写C/C++代码时也具有高亮语法和自动补全代码的功能。

(3) Cygwin——由于交叉编译NDK本地代码需要用到UNIX环境,因此需要借助于Cygwin或者MinGW之类的提供Windows下UNIX环境的应用程序。

(4) Eclipse的Sequoyah插件——该插件主要是方便所有移动应用程序的开发者的本地代码开发工作,利用Sequoyah插件可以方便地为现有的Eclipse项目添加对本地代码的

支持。具体使用方法将在 10.2.5 节中介绍。

(5) Android NDK：其作用见 11.1 节。

(6) Android 基础开发环境：见第 4 章，因为 NDK 并不能生成.apk 文件，一般来说它仅仅用于生成前面提到的 so 文件即动态库文件。

10.2.2 安装 Android NDK

安装 Android NDK 非常简单，仅仅需要在 Android 在线文档的 SDK 栏内找到 Native Development Tools 下的 Android NDK 条目，下载对应操作系统的 NDK 压缩包，并解压缩至任意目录即可。但是需要注意的是，NDK 的正确使用的前提是需要安装最新版本的 Android SDK，因为虽然 NDK 能够兼容较低版本，但是不能够配合过老版本的 SDK tools 正常工作。安装 NDK 所需满足的具体要求如下：

(1) Android SDK。

要求安装完整的 Android SDK(包括其依赖的所有组件)且版本高于 1.5。

(2) 支持的操作系统。

- Windows XP(32 位) or Vista/7(32 或 64 位)。
- Mac OS X 10.4.8 或更新版本(x86 版本)。
- Linux(32 或 64 位；例如 Ubuntu 8.04，或者其他使用 GLibc 2.7 以上版本的 Linux 发行版)。

(3) 其他必需的开发工具。

- GNU Make 3.81 及以上版本。
- 最新版本的 GNU Awk 或 Nawk。
- 对于 Windows 操作系统需要使用 Cygwin 1.7 及以上版本。

确保开发所使用的计算机满足如上要求之后，可以通过如下步骤来安装 NDK：

(1) 在 NDK 官方下载页面 http://developer.android.com/sdk/ndk/index.html 下载最新版本的 NDK 安装包，如图 10-1 所示。

Platform	**Package**	**Size**	**MD5 Checksum**
Windows	android-ndk-r6b-windows.zip	67670219 bytes	f496b48fffb6d341303de170a081b812
Mac OS X(intel)	android-ndk-r6b-darwin-x86.tar.bz2	52798843 bytes	65f2589ac1b08aabe3183f9ed1a8ce8e
Linux 32/64-bit(x86)	android-ndk-r6b-linux-x86.tar.bz2	46532436 bytes	309f35e49b64313cfb20ac428df4cec2

图 10-1 下载 NDK 安装包

(2) 解压下载好的压缩包到任意目录，解压后的文件夹名称为 android-ndk-<版本号>，可以按个人的偏好来设置这个文件夹的名称，在本书的其他地方将以“<ndk>”来代表 NDK 的安装路径。

10.2.3 安装 Cygwin

Cygwin 是一个在 Windows 平台上运行的 UNIX 模拟环境，是 Cygnus Solutions 公司开发的自由软件。它对于学习 UNIX/Linux 操作环境，或者从 UNIX 到 Windows 的应用

程序移植，或者进行某些特殊的开发工作，尤其是使用 gnu 工具集在 Windows 上进行嵌入式系统开发时，非常有用。Cygwin 的安装步骤如下。

(1) 进入 Cygwin 的官方主页 http://www.cygwin.com，在网站首页左方的导航栏里点击"Install Cygwin"进入安装指引页面。在该页面可以找到 Cygwin 的安装程序 setup.exe 的下载链接，点击进行下载。

(2) 下载完毕后运行 setup.exe 开始安装 Cygwin。安装过程相对简单，根据安装向导依次选择 Install from Internet，选择 Cygwin 安装的根目录，选择存放安装包的路径，选择使用的网络连接(一般情况下选择直接连接，如果使用了代理服务器进行上网，需要在此做相应的设置)，之后安装向导会在网络上搜寻可用的下载连接，对于中国大陆地区的用户可以选择由网易公司提供的安装镜像连接 http://mirrors.163.com，单击"下一步"按钮之后会开始下载安装所必需的一些文件。

(3) 之后进入到比较重要的一步，就是选择需要安装的一些 package，界面如图 10-2 所示。

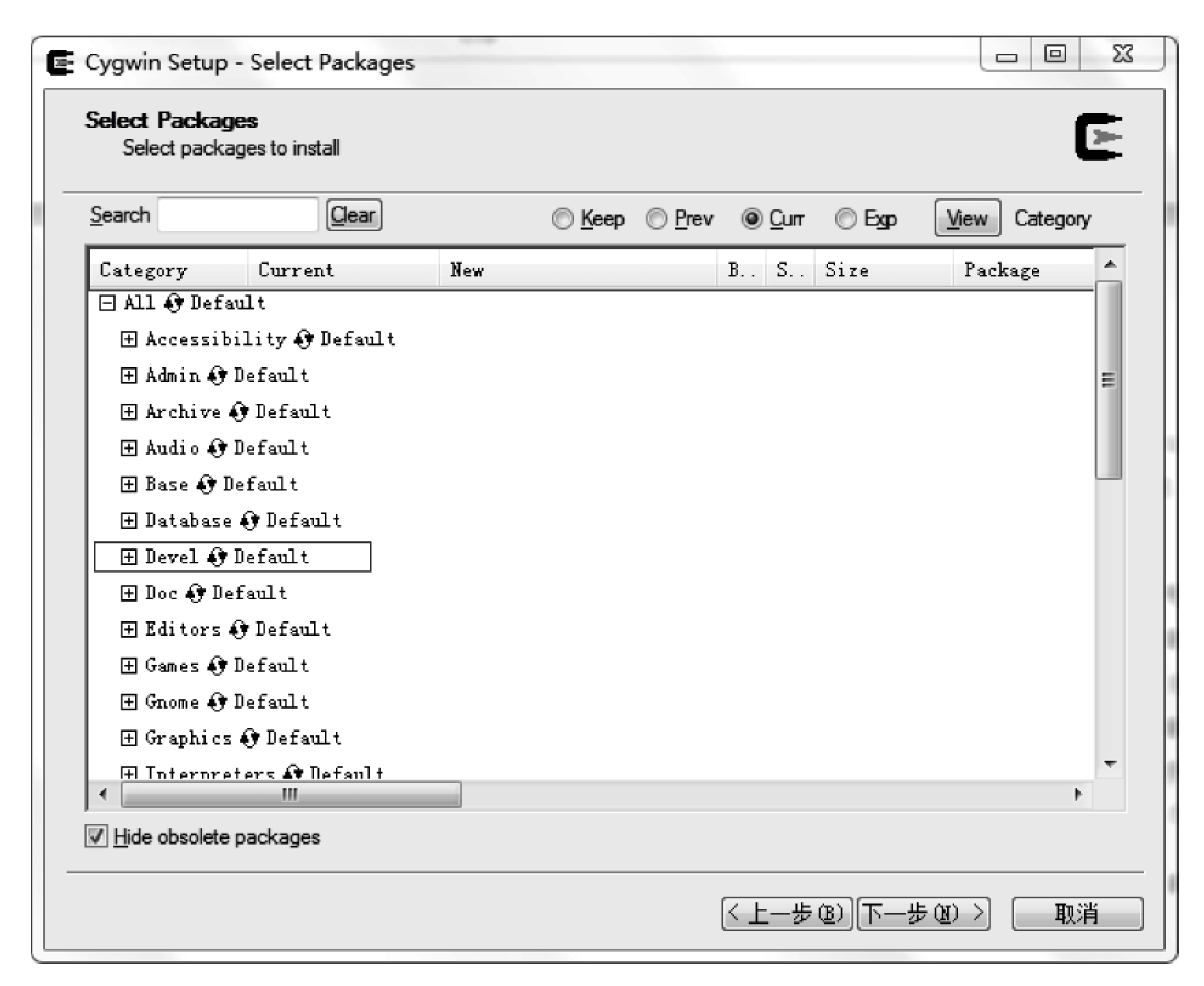


图 10-2 Cygwin 安装界面——选择软件包

这里先介绍如何去选择安装 make 软件包，图 10-3 中的选项就是 make 所在的分支，点击 Devel 前面的"+"，打开树形选项，在列表中找到 package 名称为"make: The GNU version of the 'make' utility"的一项，点击 skip 将其改变为版本号，如"3.81-2"，此时会发现名称为 Bin 的一列的 n/a 已经变为了一个选中的方框，这时就意味着选中了安装 make，

如图 10-3 所示。

3.81-2 ☒ n/a 350k make: The GNU version of the 'make' utility

图 10-3 选中 make 软件包

以同样的方法选择其他 package 进行安装，除了 make 这里推荐需要安装的一些包，这些包在之后的开发工作中会用到 autoconf2.1、automake1.10、binutils、gawk、gcc-core、gcc-g++、gcc4-core、gcc4-g++、gdb、pcre、pcre-devel。为了方便寻找这些包，可以单击界面右上角的 View 按钮，将列表切换为 Full(默认的是 Category)，选择完成后单击“下一步”按钮即开始安装。值得说明的是，Cygwin 提供的这个安装向导程序可以在之后的时间运行来安装其他的需要的 package 而不需要重复安装 Cygwin 及其他的 package。安装完成后在运行的 Cygwin 环境中可以通过 make-v 和 gcc-v 来确认它们是否已经正确安装，如图 10-4 所示。



图 10-4 检查 make 是否正确安装

通过以上的一些步骤就基本完成了 Cygwin 的安装，利用 Cygwin 提供的 UNIX 环境，就能够使用 NDK 的编译工具编译本地代码了，可以利用安装好的环境来编译并运行 NDK 自带的 samples。下面就来介绍如何编译并运行一个包含有本地代码的 sample——hello-jni。

hello-jni 示例在<ndk>/samples 下，这个项目是熟悉 NDK 的最简单的一个项目，类似于 Hello World。这个项目完成的功能就是通过 JNI 调用一个本地方法在屏幕上显示一段字符串，本地方法使用 C 语言代码实现，经过 NDK 编译形成动态库(.so)供 jni 调用。执行这个项目的步骤如下：

(1) 在 eclipse 中通过 Create project from existing source 方法建立起 hello-jni 项目。需要注意的是在选择 API level 时需要选择 1.5 或更高的版本。如图 10-5 所示。

(2) 使用 NDK 根目录下的 ndk-build 文件编译本地代码，在一般的 Linux 环境下只需要如下步骤：

首先，将目录切换至第(1)步中建立的项目根目录：

```
cd <ndk>/samples/hello-jni      //<ndk>即为解压目录
```

然后，在 UNIX 环境下执行<ndk>/ndk-build 即可完成编译。由于现在是在 Windows 环境下来进行编译，因此需要借助于 Cygwin 来完成，为此运行前面安装好的 Cygwin。在 Cygwin 运行起来后，会得到一个类似于 Linux 下终端的界面，在这个界面里就可以使用

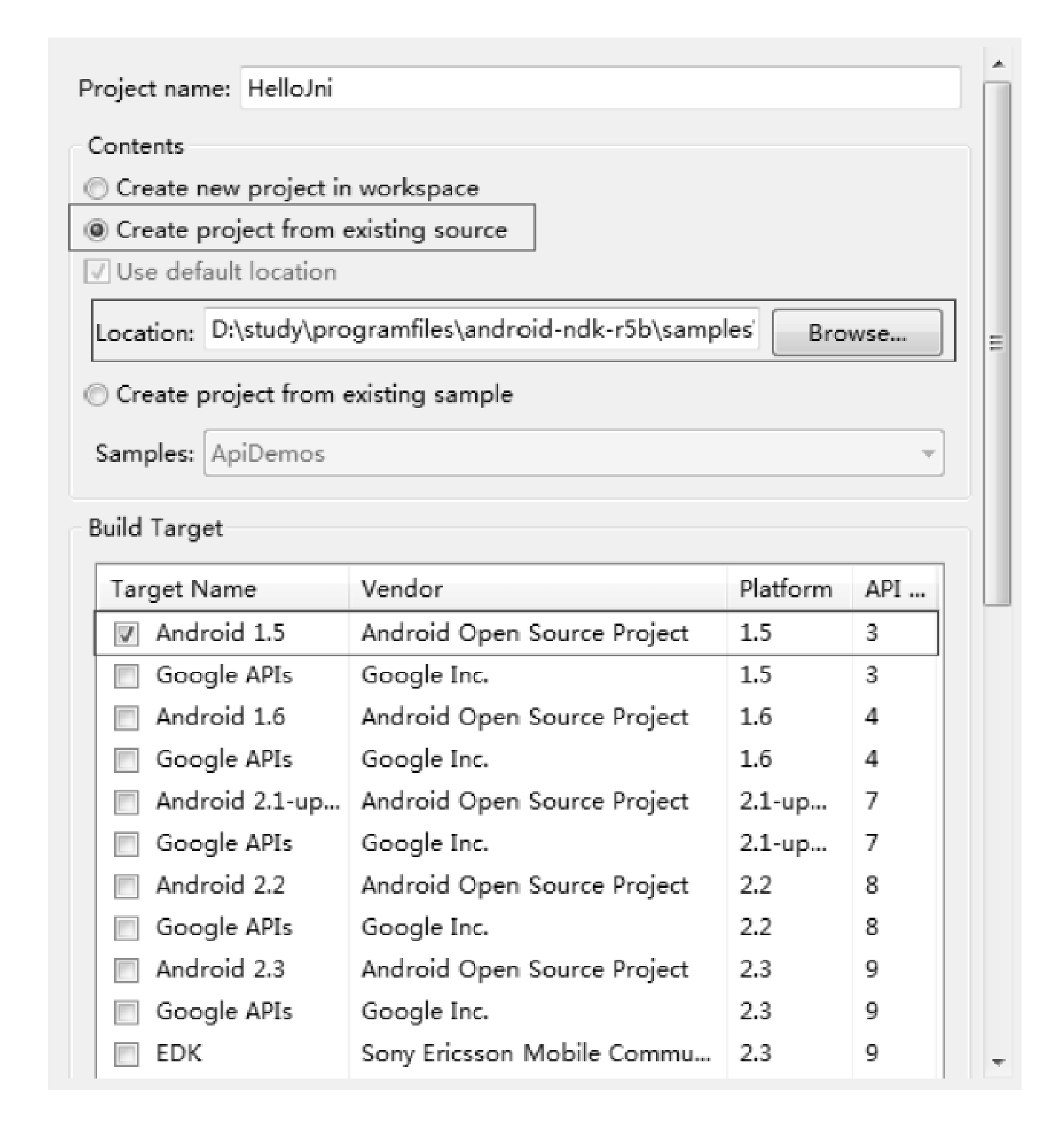


图 10-5 建立 Hello-jni 项目

Linux 的命令进行相关操作了，如何在 Cygwin 环境下找到对应的 Windows 下的文件或目录呢？很简单，Cygwin 在根目录下有一个名为 cygdrive 的目录，这个目录下就虚拟地挂载了 Windows 下所有可以访问的磁盘，如图 10-6 所示。

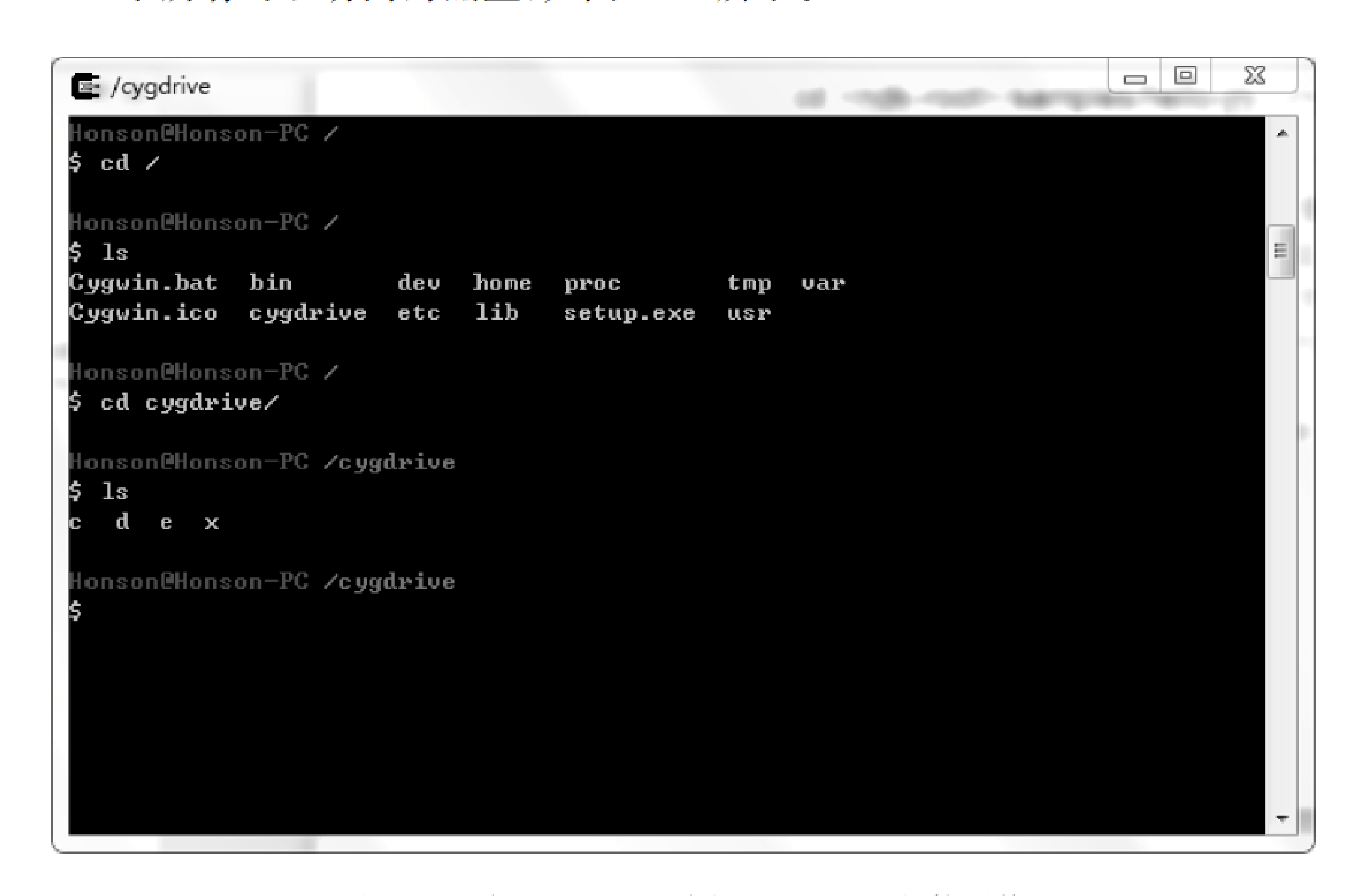


图 10-6 在 Cygwin 下访问 Windows 文件系统

（3）通过 Cygwin 进入到对应目录在执行＜ndk ＞/ndk-build 命令即可完成编译。过程如图 10-7 所示。

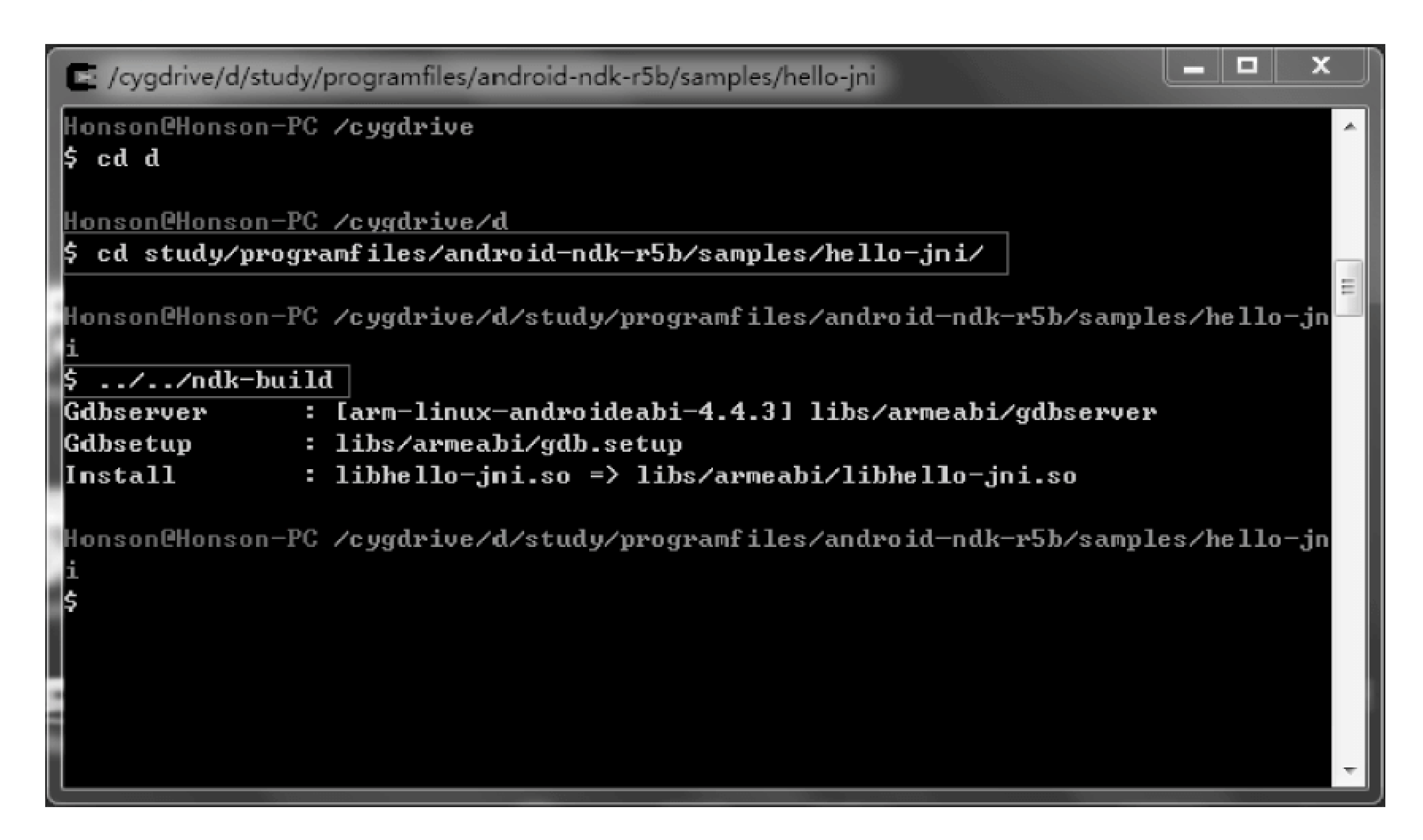


图 10-7　执行 ndk-build 命令进行编译

编译完成后在 eclipse 中对项目进行刷新后可以发现增加了如图 10-8 所示的两个目录。

这就是编译后生成的目标文件，现在就可以像运行普通 Java 编写的应用一样运行该例子了。运行结果如图 10-9 所示，对于具体的原理将在后面的章节中进行讨论。



图 10-8　编译后的项目文件　　　　图 10-9　hello-Jni 运行结果

10.2.4　安装 Eclipse 下 C/C++ 开发工具

CDT 插件的官方下载点为 http://www.eclipse.org/cdt/downloads.php，在该页面上有适用于各个不同版本 Eclipse 的 CDT，找到正确的版本，例如适用于 Eclipse Indigo 版本的一栏如图 10-10 所示。

如图 10-10 所示，首先是一个 Eclipse C/C++ IDE Indigo SR-1 的下载链接，该链接是用于下载自带集成了 CDT 插件的 Eclipse 开发环境，如果当前计算机上没有安装任一版本的 Eclipse，则可以选择下载安装此版本 Eclipse，则无须再另外安装 CDT；如果已经安装了 Eclipse Indigo，则可以使用第二个连接 p2 software repository 在 Eclipse 的 Help 菜单下选


CDT 8.0.1 for Eclipse Indigo

Eclipse package: Eclipse C/C++ IDE Indigo SR-1.

p2 software repository: http://download.eclipse.org/tools/cdt/releases/indigo.

The git repos have been tagged with the CDT_8_0_1 tag. You can download the source from the web interface.

Archived p2 repos:

- cdt-master-8.0.1.zip
- cdt-master-8.0.0.zip

图 10-10 CDT 资源

择 Install New Software 并复制如上所述链接进行在线安装，选中 Group items by category，会按分类出现供安装的组件，此处不需要安装所有的组件，推荐选中 CDT Main Features 分类并选中 CDT Optional Features 下的 C/C++ Development Platform、C/C++ DSF GDB Debugger Integration、C/C++ GCC Cross Compiler Support、C/C++ GNU Toolchain Build Support、C/C++ GNU Toolchain Debug Support、Eclipse Debugger for C/C++、Miscellaneous C/C++ Utilities 这些组件，其他组件可以在需要用的时候再进行安装，如图 10-11 所示。

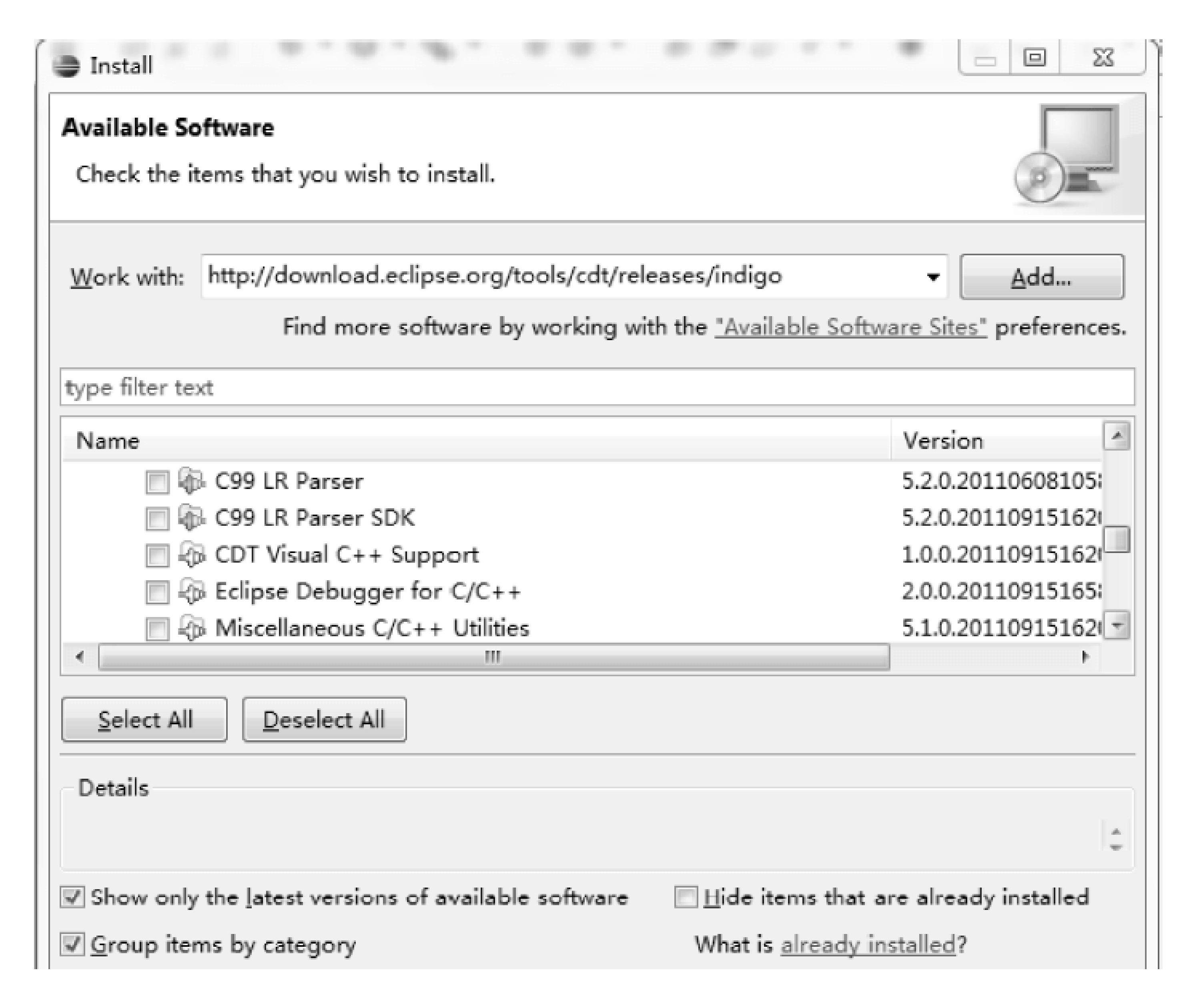


图 10-11 CDT 安装

待安装完成后，如果能在新建项目的选项中（New→Other）发现 C/C++ 一类，即说明 CDT 安装成功。

如果在线安装的方法由于网络原因或者其他原因不能够成功完成，则可以通过下载离线安装包的方式进行安装，首先需要通过如图 10-10 中最下方的链接下载 CDT 安装包，例如目前最新的 8.0.1 版本，下载到本地后，在如图 10-11 所示的界面中单击地址栏右方的 Add 按钮，然后单击 Archive 按钮并定位到刚下载的 cdt-master-8.0.1.zip 压缩包，再进行

安装即可。

如图 10-12 所示，对于现有的 Java 或 Android 项目可以通过列表下的第四个选项"Convert to a C/C++ Project(Add C/C++ Nature)"来为项目添加 C/C++相关支持。为项目添加了 C/C++特性之后将 Eclipse 的 Perspective 切换到 C/C++下时，可以看到项目结构下会多出 Includes 目录，该目录下包含了一系列用 C/C++开发时会用到的头文件，如果是为了配合使用 NDK 进行 C/C++开发，则应该在 Includes 内添加 NDK 提供的一些头文件，这部分过程可由 10.2.5 节介绍的 Sequoyah 插件自动实现，因此这里不做介绍。对该项目执行 Build 后，项目架构中还会增加 Binaries 目录，如图 10-13 所示。

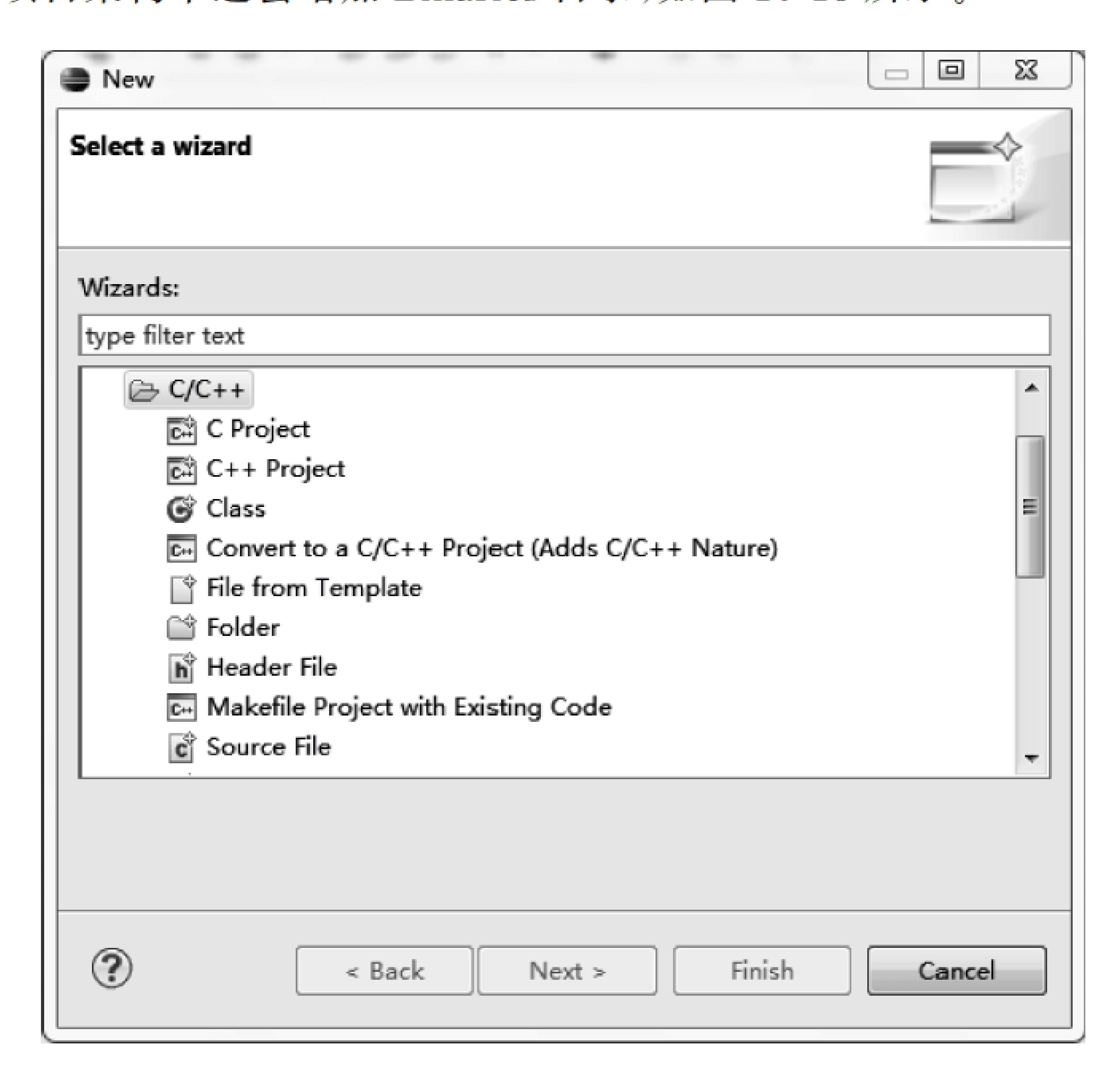


图 10-12　CDT 安装成功

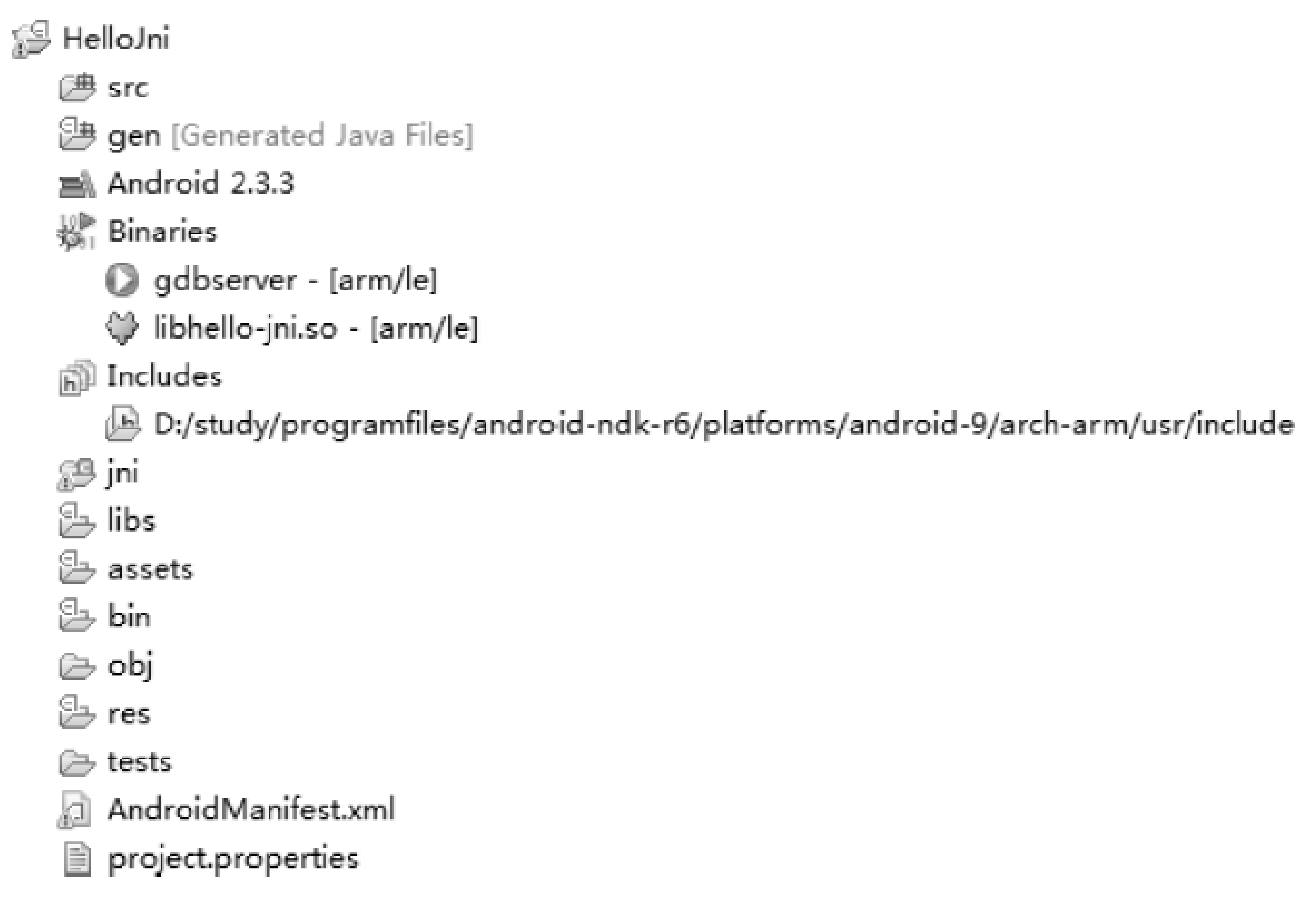


图 10-13　含 C/C++特性的项目结构

10.2.5　安装 Eclipse 下 Sequoyah 插件

Sequoyah 插件的官方下载点为 http://www.eclipse.org/sequoyah/downloads/，在该网页上同样提供了用于在线安装的 update site 地址以及安装包的下载地址，需要注意的是，在安装界面要确认 Group items by category 复选框处于选中状态，否则可能出现列表为空(There are no categorized items)的情况。如图 10-14 所示，全部选中列出的安装包并完成安装。

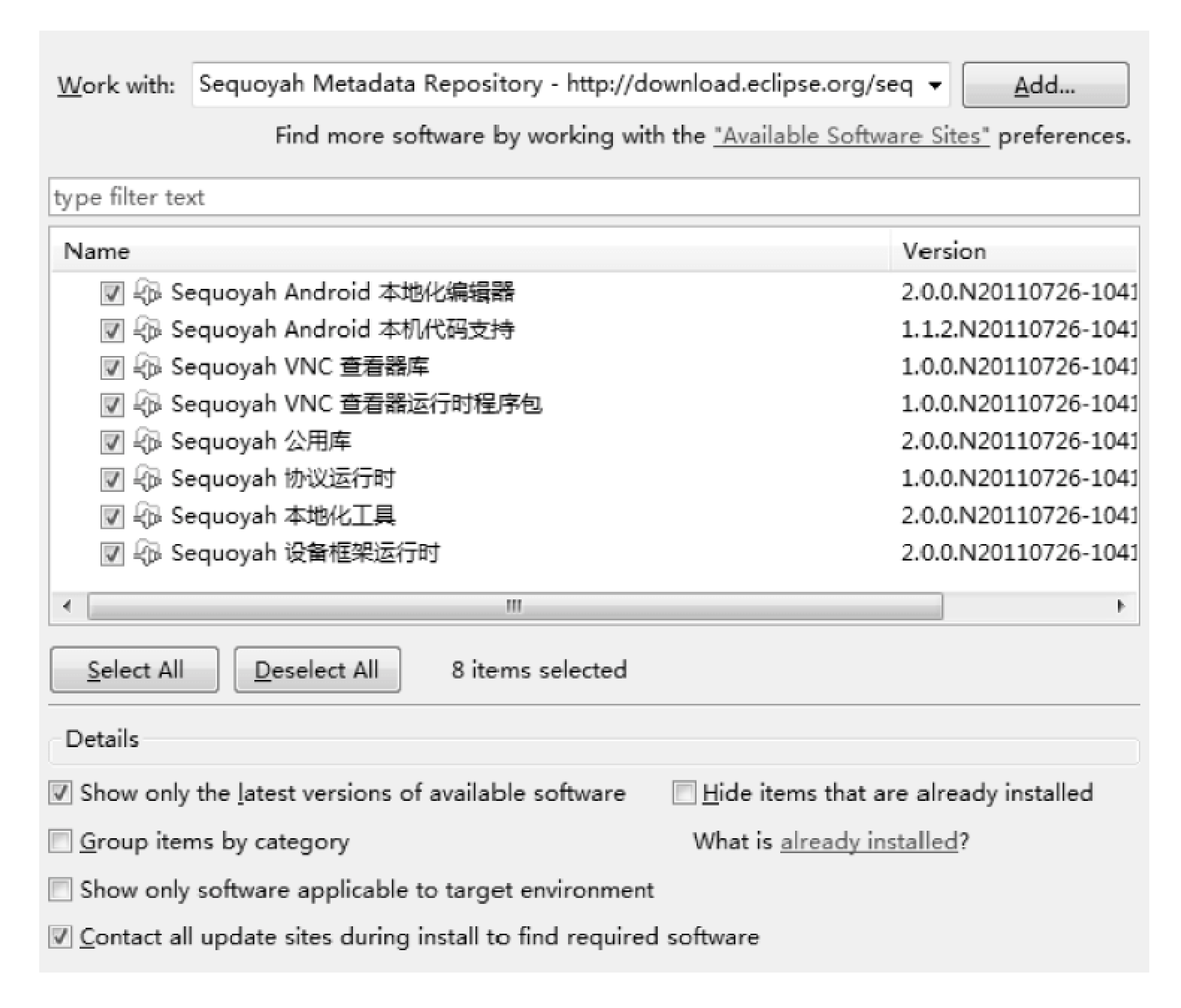


图 10-14　安装 Sequoyah

Sequoyah 安装完成后，右击任意的 Android Project，会发现在 Android Tools 中多出了一个 Add Native Support 选项(如图 10-15 所示)，单击后会弹出一个简单的设置界面，如图 10-16 所示，在项目栏里填写需要添加本地支持的项目名称，也可以通过单击“浏览”按钮

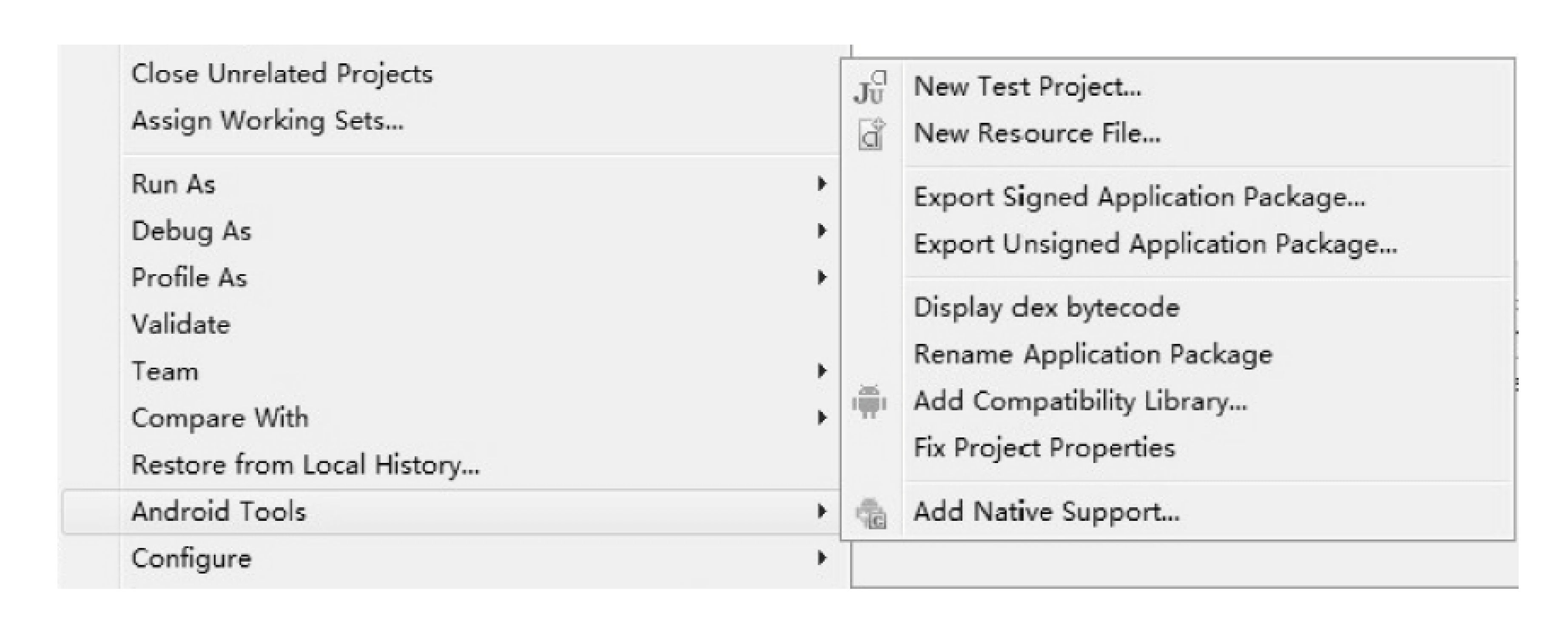


图 10-15　Sequoyah 安装成功

定位项目；在 NDK 位置栏中是计算机中安装 NDK 的根目录，即<ndk>，设置该路径的方法为：Window→Preferences→Android→“本机开发”(如图 10-17 所示)。

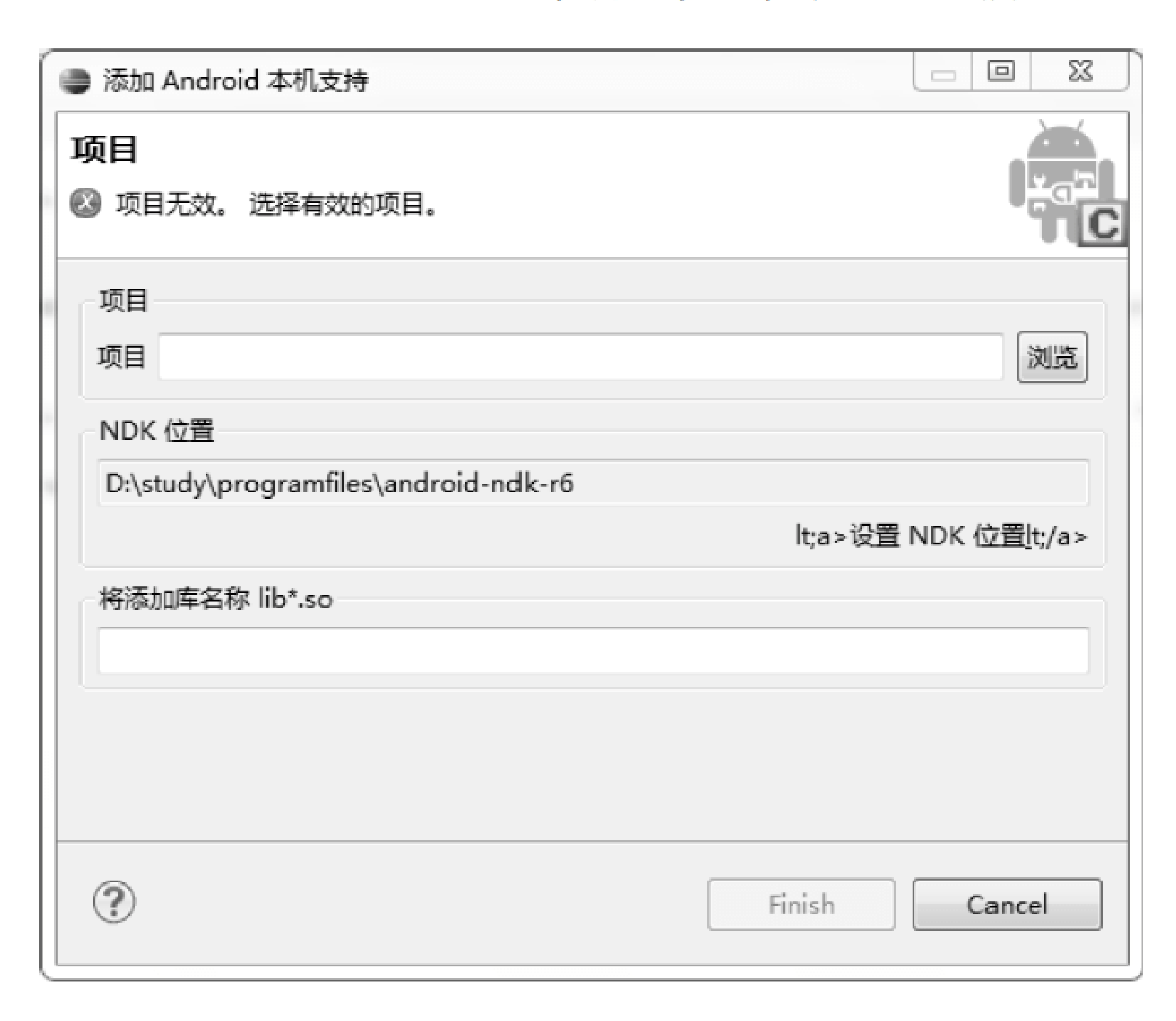


图 10-16　添加 Android 本机支持

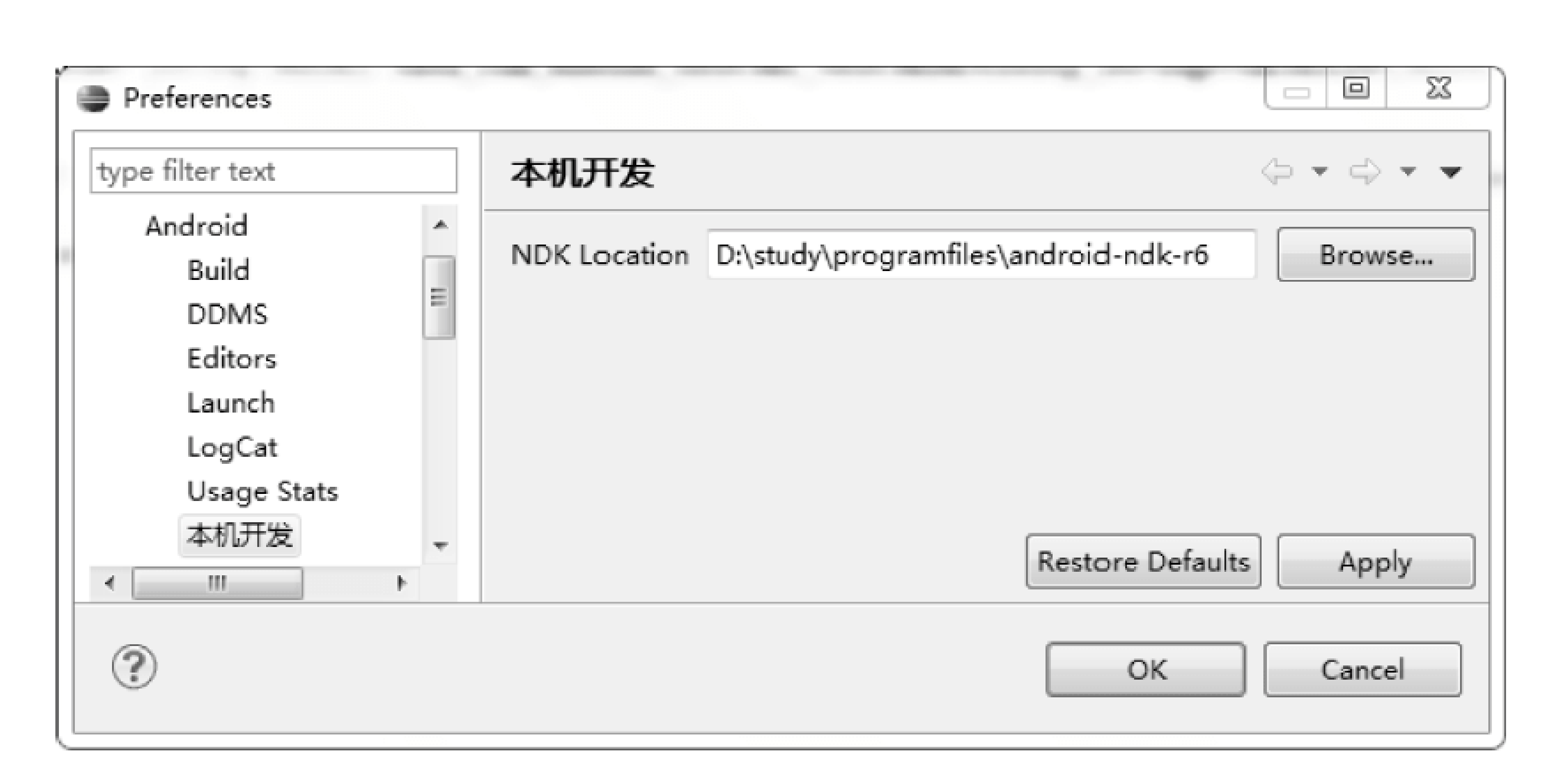


图 10-17　设置 NDK 安装路径

在将添加库名称 lib＊.so 一栏中填写需要生成库的名称，例如对于 hello-jni 项目的库名称即为 libhello-jni.so，在此处填写 hello-jni 即可。

另外，Sequoyah 提供了详细的帮助文档，遇到问题时可以通过 Eclipse 的 Help 菜单下的 Help Contents 命令进行查看。

10.2.6　验证开发环境：NDK 入门示例

为了验证经过如上步骤所搭建起来的 NDK 开发环境的可使用性，本节将演示如何从零开始建立一个包含有 JNI 调用本地代码方法的 Android Project。首先，新建一个

Android 项目，命名为 TestEnv，单击 Next 按钮，选择一个 SDK 的版本。由于 Sequoyah 要求是 API Level 在 9 及以上，因此选择一个 API Level 大于 9 的 SDK 版本，再单击 Next 按钮，如图 10-18 和图 10-19 所示。

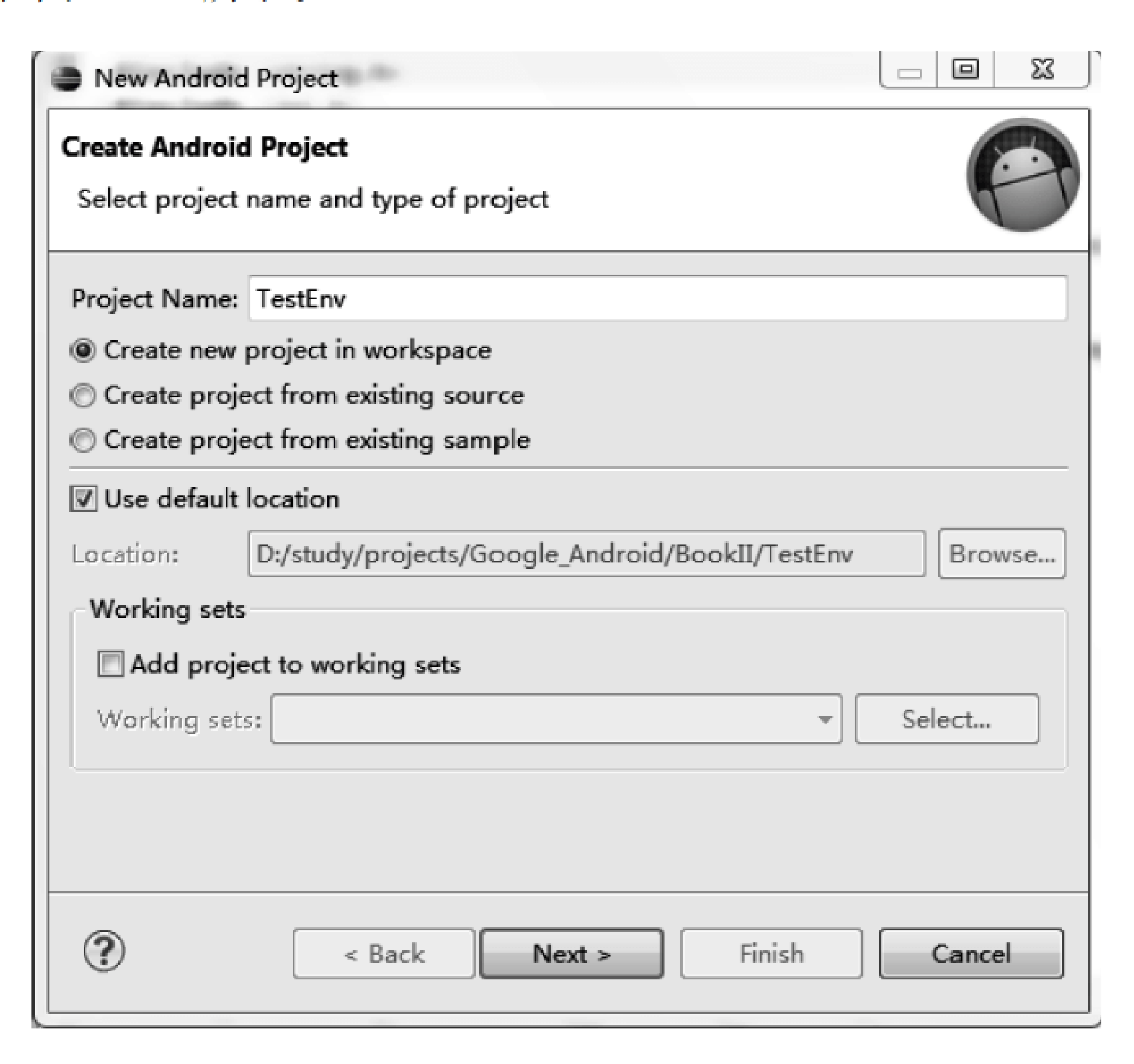


图 10-18 选择 SDK

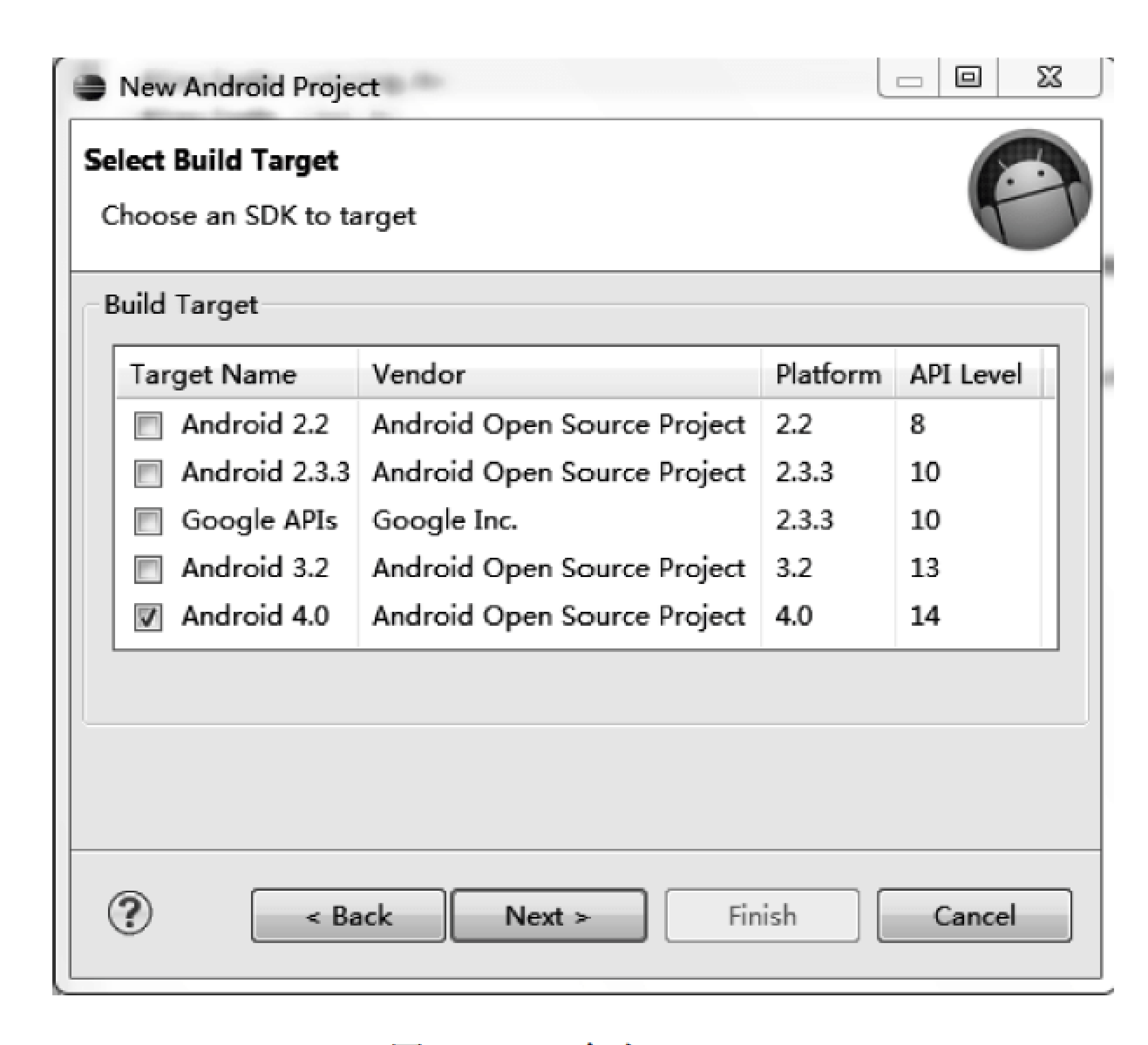


图 10-19 命名 Project

之后需要输入 Package Name 即包名（如图 10-20 所示），要求包的名称具有唯一性，因为它有标识应用程序的作用，通常使用域名加上标识符的形式，这里作为示例使用

com. android. example 作为包名，单击 Finish 按钮完成创建，得到新的 Android Project 内容，如图 10-21 所示。

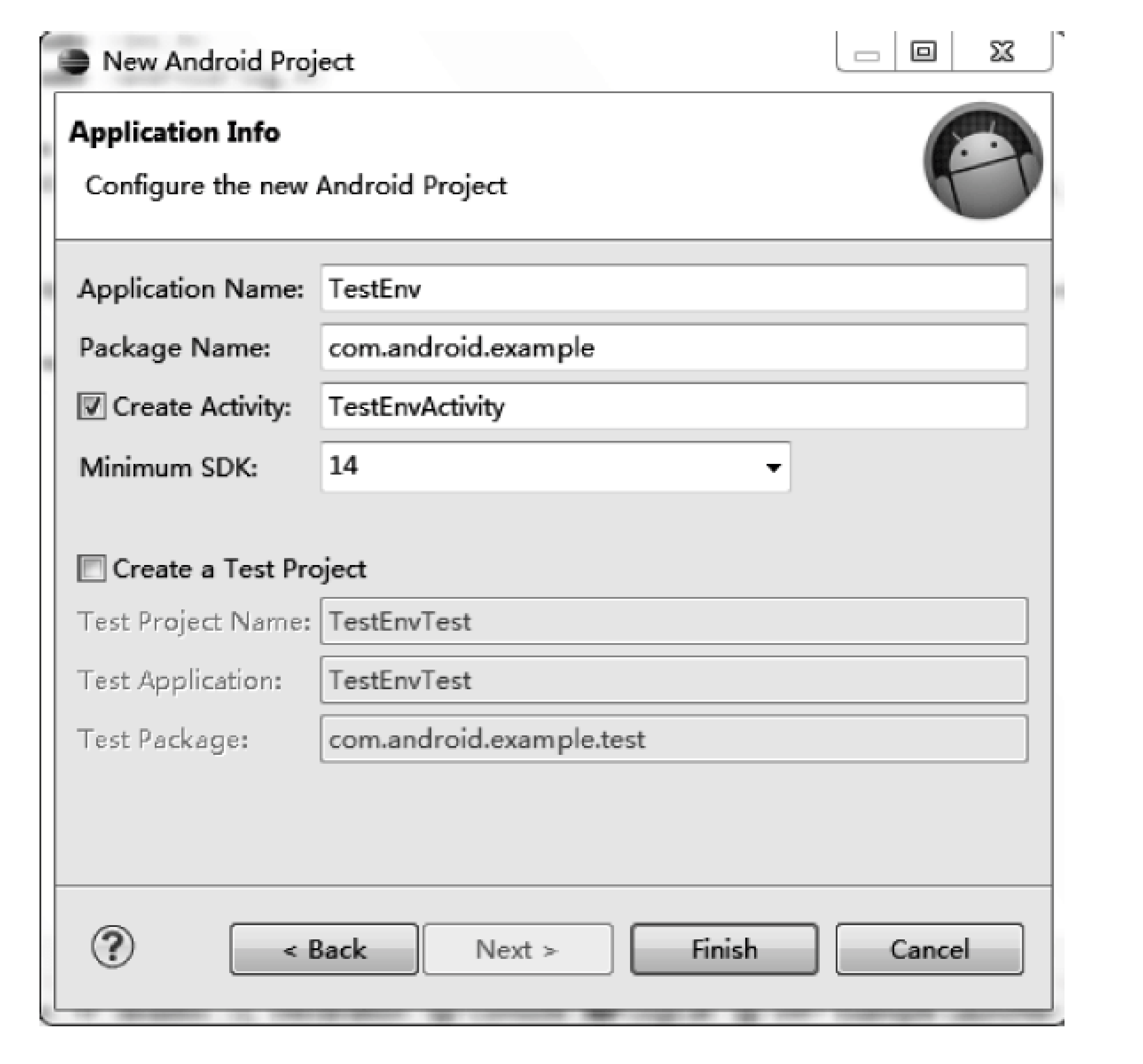


图 10-20　命名包名

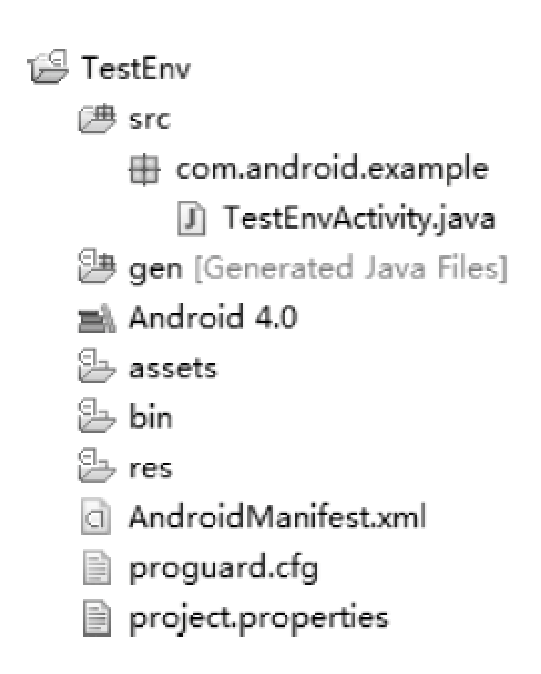


图 10-21　项目建立完成

新建好一个空的项目后，在 Project Explorer 中右击 TestEnv 项目，并在 Android Tools 中选择 Add Native Support 命令，弹出如图 10-22 所示的界面，保持默认设置。单击 Finish 按钮完成后项目的结构将变成如图 10-23 所示，可以看到多了一个名为 jni 的目录，这个目录便是 Sequoyah 自动生成的用于存放 C/C++源代码的目录，该目录下存在两个文件。Android. mk 是用于描述代码的编译规则的，相当于 makefile，它的内容为：



图 10-22　添加本地支持

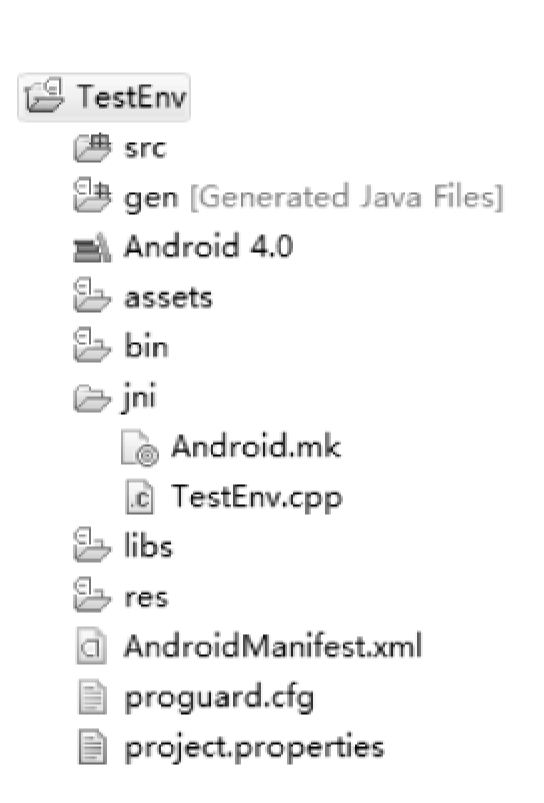


图 10-23　添加本地支持后的结构

```
01 LOCAL_PATH := $(call my-dir)
02 include $(CLEAR_VARS)
03 LOCAL_MODULE := TestEnv
04 ### Add all source file names to be included in lib separated by a whitespace
05 LOCAL_SRC_FILES := TestEnv.cpp
06 include $(BUILD_SHARED_LIBRARY)
```

其中第 01 行的内容是对 LOCAL_PATH 变量进行赋值,每个 Android.mk 文件都必须以此操作为第一句,该语句的作用是在项目的目录树中定位源代码文件,NDK 的编译系统根据这个变量来寻找所需的源代码文件。在此处用到了由 NDK 编译器所提供的一个宏函数 my-dir,它能够返回当前目录,在此处即是包含 C++代码的 jni 目录。

第 02 行 include $(CLEAR_VARS),其中的 CLEAR_VARS 变量是由 NDK 编译器提供的变量,该变量指向了一个特定的 GNU Makefile,这个 Makefile 的作用是重置所有除 LOCAL_PATH 外的其他所有 LOCAL_XXX 形式的变量,由于所有控制编译的文件都是在一个唯一的 GNU Make 执行环境中进行解析的,因此必须在开始新的编译前正确处理所有的全局变量。

第 03 行 LOCAL_MODULE := TestEnv,LOCAL_MODULE 变量是用于指定所有需要编译的模板,因此这一行也是必须具备的,这个变量的值必须是唯一的,且不包含空格字符,NDK 在生成模块时会自动为该变量值添加前缀"lib"和后缀".so",因此在本例中将生成名为 libTestEnv.so 的模块,这个模块名将在 Java 代码中用于加载模块的操作,在后面将会看到。

第 04 行(不记注释行)LOCAL_SRC_FILES := TestEnv.cpp,这一行相对简单,LOCAL_SRC_FILES 即用于指定源代码文件的变量,该例中只存在由 Sequoyah 生成的一个源文件 TestEnv.cpp;如果存在多个源文件,则在多个文件名中用空格分开。在列出源文件时不需要把头文件或者 included 的文件列出来,因为编译器会自动计算这些依赖性,仅列出源文件即可。

第 05 行 include $(BUILD_SHARED_LIBRARY),BUILD_SHARED_LIBRARY 也是由编译器提供的指向一个 GNU Makefile 文件的变量,该 Makefile 的作用是搜集所有的从最近一条"include $(CLEAR_VARS)"语句之后的所有 LOCAL_XXX 形式的变量信息,从而分析得出应该如何来进行编译并完成相应的编译构建工作。与此相关的还有 BUILD_STATIC_LIBRARY 变量用于构建一个静态的库,两者的区别是:只有 shared library 会被复制到应用程序的安装包,而 static library 也可用于生成 shared library。

TestEnv.cpp 文件是 C++源代码文件,这个默认的文件是空的,只包含了两条 include 语句。

```
#include <string.h>
#include <jni.h>
```

为了通过使用 JNI 调用来验证开发环境,修改 TestEnv.cpp 代码如下:

```
#include <string.h>
#include <jni.h>
```

```
extern "C" {
    JNIEXPORT jstring JNICALL Java_com_android_example_TestEnvActivity_stringFromJNICPP
(JNIEnv * env, jobject obj);
};

JNIEXPORT jstring JNICALL Java_com_android_example_TestEnvActivity_stringFromJNICPP
    (JNIEnv * env, jobject obj)
{
     return env->NewStringUTF("Hello From CPP");
}
```

如上面的代码所示，其功能是返回字符串“Hello From CPP”。由于 NDK 主要是配合 C 语言开发，因此前面一段添加了 extern C 关键字，用于告诉编译器其内部包含的函数是使用 C 语言编写的。注意函数名的命名特点，这个命名是与 Java 代码中的包名、类名及声明的方法名相关，具体命名规则遵循 JNI 的规范（将在 11.3.1 节中对此本地方法名的解析规则进行进一步的说明）。简言之，Java_com_android_example_TestEnvActivity_stringFromJNICPP 说明该方法对应在 Java 代码的 com.andorid.example 包内的 TestEnvActivity 类中声明，方法名称为 stringFromJNICPP。保存对 TestEnv.cpp 的修改后，可能发现 Eclipse 会报错，提示找不到与 JNI 相关的一些定义，这时需要通过右键快捷菜单修改 TestEnv 项目的属性，即右击项目，选择 Properties→C/C++ Gerneral→Paths and Symbols→Includes 选项卡下的 GNU C 选项，单击 Add 按钮并选择 File system 添加<ndk>\platforms\android-9\arch-arm\usr\include 后重新 Build 即可解决，如图 10-24 所示。

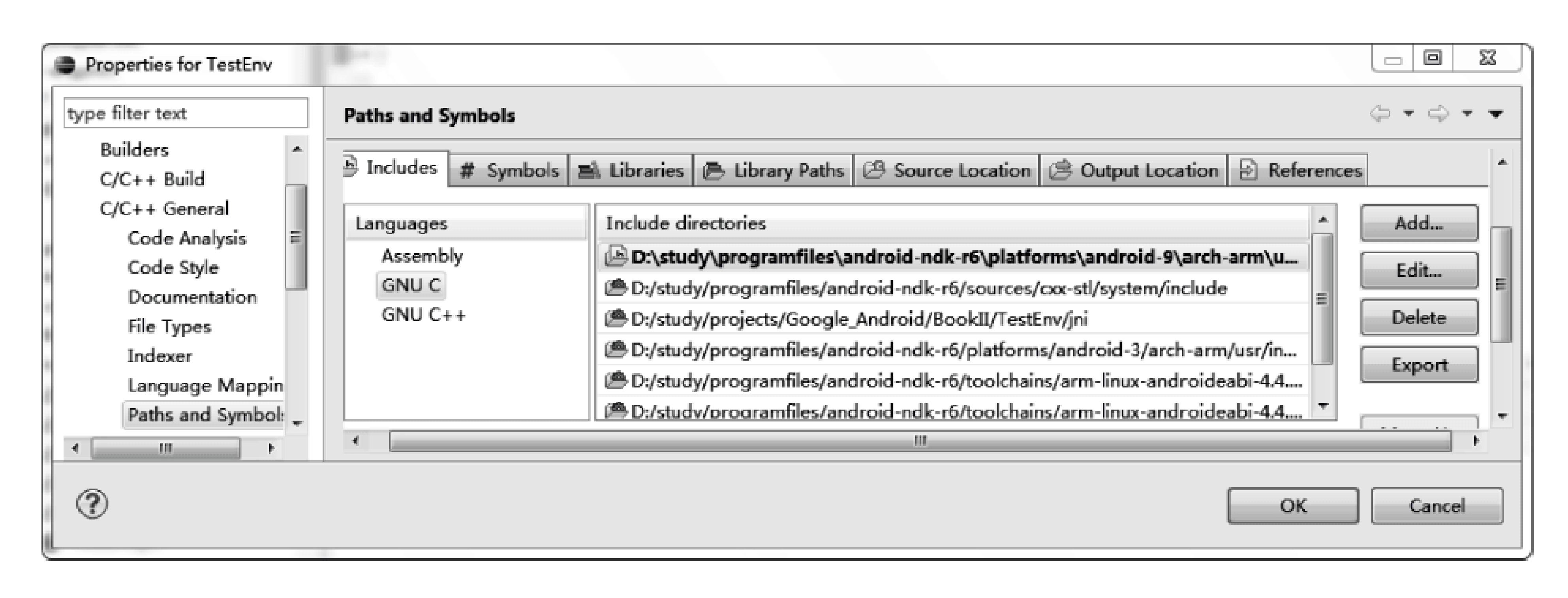


图 10-24 添加 Includes

之后需要修改项目的 TestEnvActivity.java 代码和 main.xml（TestEnvActivity 默认的布局文件）文件，使程序通过 JNI 调用本地代码提供的方法返回字符串，并显示到 Activity 的一个 TextView 中。首先在 main.xml 中添加一个 TextView 控件，代码如下：

```
<?xml version = "1.0" encoding = "utf - 8"?>
<LinearLayout xmlns:android = "http://schemas.android.com/apk/res/android"
    android:layout_width = "fill_parent"
    android:layout_height = "fill_parent"
    android:orientation = "vertical" >
    < TextView
```

```
            android:id = "@+ id/myTextField"
            android:layout_width = "fill_parent"
            android:layout_height = "wrap_content"
            android:text = "@string/hello" />
</LinearLayout>
```

之后再修改 TestEnvActivity.java，如下：

```
package com.android.example;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class TestEnvActivity extends Activity {
    /** Called when the activity is first created.  */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView myTextField = (TextView)findViewById(R.id.myTextField);
        myTextField.setText(stringFromJNICPP());
    }

    public native String stringFromJNICPP();

    static {
        System.loadLibrary("TestEnv");
    }
}
```

保存文件后，单击 Eclipse→Project 选项，取消选中 Build Automatically 选项，然后右击 TestEnv 项目在弹出的快捷菜单中选择 Build Project 命令，对项目进行构建并生成应用程序，如图 10-25 所示。这一步需要观察 Eclipse Console(如果没有该视图，则通过 Window→Show View→Console 命令显示)的输出，正常情况下应该如图 10-26 所示，如果 Build 工具没有配置正确，则可能输出错误信息，如图 10-27 所示。

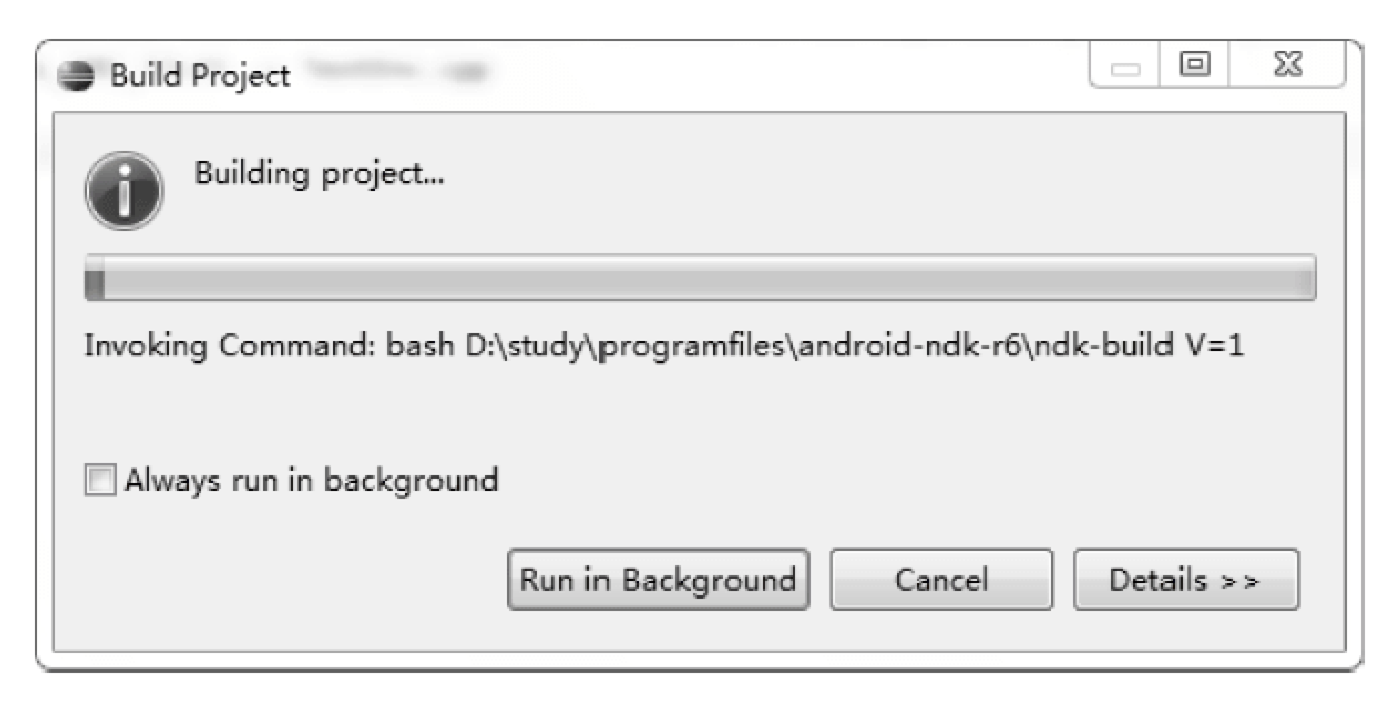


图 10-25　Build 过程

```
Problems  @ Javadoc  Declaration  Console  LogCat  SWT Example Launcher  Call
CDT Build Console [TestEnv]
D:/study/projects/Google_Android/BookII/TestEnv/obj/local/armeabi/libTestEnv.so
Install        : libTestEnv.so => libs/armeabi/libTestEnv.so
mkdir -p /cygdrive/d/study/projects/Google_Android/BookII/TestEnv/libs/armeabi
install -p /cygdrive/d/study/projects/Google_Android/BookII/TestEnv/obj/local/armeabi/libTes
/cygdrive/d/study/projects/Google_Android/BookII/TestEnv/libs/armeabi/libTestEnv.so
/cygdrive/d/study/programfiles/android-ndk-r6/toolchains/arm-linux-androideabi-4.4.3/prebuil
--strip-unneeded D:/study/projects/Google_Android/BookII/TestEnv/libs/armeabi/libTestEnv.so

**** Build Finished ****
```

图 10-26 Build 成功

```
Problems  Tasks  Console  Properties
CDT Build Console [TestEnv]

**** Build of configuration Default for project TestEnv ****

ndk-build V=1

Error: Cannot run program "ndk-build": Launching failed

**** Build Finished ****
```

图 10-27 Build 失败

出现这个错误的原因是没有正确运行 ndk-build，因为 ndk-build 只能够在 UNIX 环境下执行，而当前系统环境是 Windows，Build 默认的设置可能是直接运行 ndk-build，因此出现了这个错误，修改的方法是右击 TestEnv 项目，在弹出的快捷菜单中选择 Properties→C/C++ Build 命令，在 Builder Settings 选项卡中，取消选中 Use default build command 选项，并在 Build Command 中填入：bash ＜ndk＞\ndk-build，其中的＜ndk＞需要替换为安装 ndk 的根目录，如图 10-28 所示。如果出现的错误是“Cannot run program "bash": Launching failed”，则是由于 Windows 环境变量中没有加入 Cygwin 的 bin 目录(bash 即在该目录下，以 bash 命令运行 ndk-build 即等效于在 UNIX 环境下运行了)，添加即可。

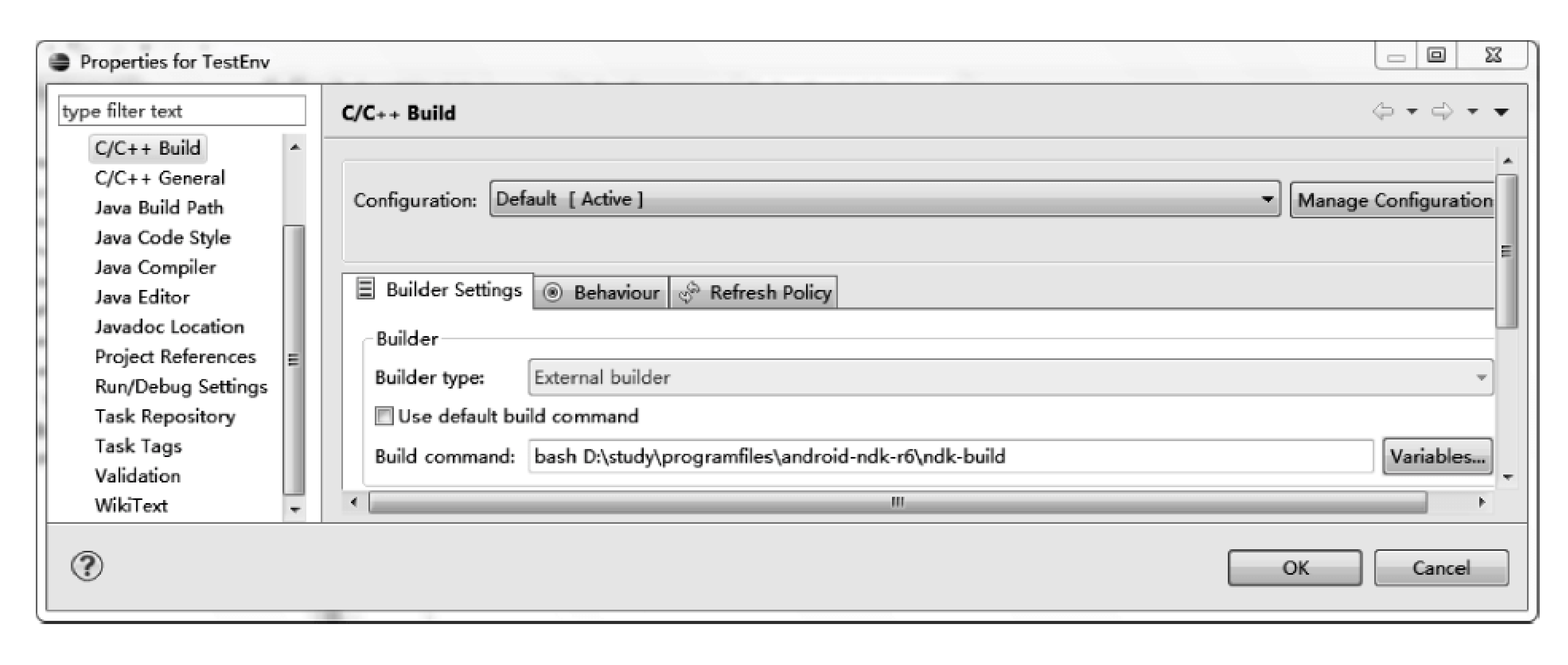


图 10-28 修改 Build Command

Build 成功后，可以发现 TestEnv 项目中增加了 Binaries，其中包含了生成的库文件 libTestEnv. so，如图 10-29 所示。

然后在模拟器上运行该项目，即可看到如图 10-30 所示的结果，至此，验证了已经正确配置了 NDK 开发环境。

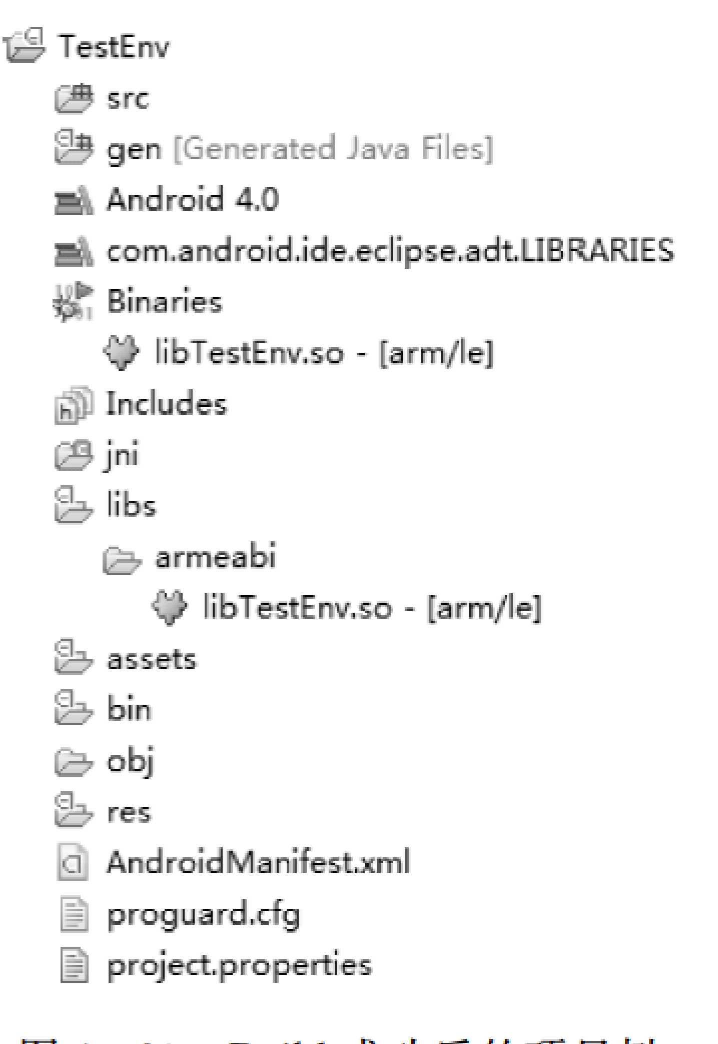




图 10-29 Build 成功后的项目树



图 10-30 运行结果

10.3 Windows 下 NDK 开发示例

通过 10.2 节介绍的搭建开发环境的过程，应该已经对如何使用 NDK 进行开发有了一个初步的了解，即：

- 首先在 Java 代码中编写带有 native 声明的方法的 Java 类。
- 然后使用 C/C++实现本地方法并通过 ndk-build 将 C/C++编写的文件生成动态链接库(lib *. so)。
- 在 Java 类中使用 loadLibrary()方法加载动态链接库，再通过 JNI 调用本地方法。

可以发现除了需要使用 C 代码来实现本地方法外，还需要的技能是对 JNI 的使用。因此，本节首先对 JNI 进行介绍，之后再完成一个 NDK 示例。

10.3.1 JNI 简介

Java 以其跨平台的特性深受人们喜爱，而由于其跨平台的目的，它和本地机器的各种内部联系变得很少，约束了它的功能。解决 Java 对本地操作的一种方法就是 JNI。JNI 是 Java Native Interface 的缩写，Java 通过 JNI 调用本地方法，而本地方法是以库文件的形式存放的(在 Windows 平台上是 DLL 文件形式，在 UNIX 机器上是 SO 文件形式)。通过调用本地的库文件的内部方法，使 Java 可以实现和本地机器的紧密联系，调用系统级的各接口方法。从 Java 1.1 开始，JNI 标准成为 Java 平台的一部分，它允许 Java 代码与用其他语言编写的代码进行交互。JNI 一开始是为了本地已编译语言尤其是 C 和 C++而设计的，但

是它并不妨碍使用其他语言，只要调用约定受支持就可以了。使用 Java 与本地已编译的代码交互，通常会丧失平台可移植性。但是，有些情况下这样做是可以接受的，甚至是必须的，比如，使用一些旧的库，与硬件、操作系统进行交互，或者为了提高程序的性能。而 JNI 标准至少能够保证本地代码能工作在任何 Java 虚拟机实现下。总而言之，使用 JNI 通常由于如下原因：

- 尽管可以完全使用 Java 来编写应用程序，但是有时单独用 Java 并不能够满足应用程序的需要。因此，程序员使用 JNI 来编写 Java 本地方法，可以处理那些不能完全用 Java 编写应用程序的情况。

而需要使用 Java 本地方法又通常由于如下几个原因：

- 标准 Java 类库不支持与平台相关的应用程序所需的功能。
- 已经拥有了一个用另一种语言编写的库，希望通过 JNI 使得 Java 代码能够访问该库，达到代码重用的效果。
- 想利用低级语言（例如汇编）来实现一小段时限代码。

既然使用了 Java 本地方法，就必然涉及本地方法对 Java 对象和数据的访问，而这些对象和数据都是存在于 Java 虚拟机环境之中的，那么本地方法的代码如何对 Java 虚拟机里的资源进行访问呢？这就需要通过调用 JNI 函数来实现，JNI 函数可以通过 JNI 接口指针来获取，这个接口指针是指针的指针，在接口中被声明为 JNIEnv *，而 JNIEnv 则在 jni.h 中被定义为：

```
typedef const struct JNINativeInterface * JNIEnv;
```

它指向一个指针数组，而指针数组中的每一个元素又指向一个接口函数。每个接口函数都处在数组的某个预定的编译量中，通过这样的组织结构来为本地代码提供 JNI 函数调用，结构如图 10-31 所示。

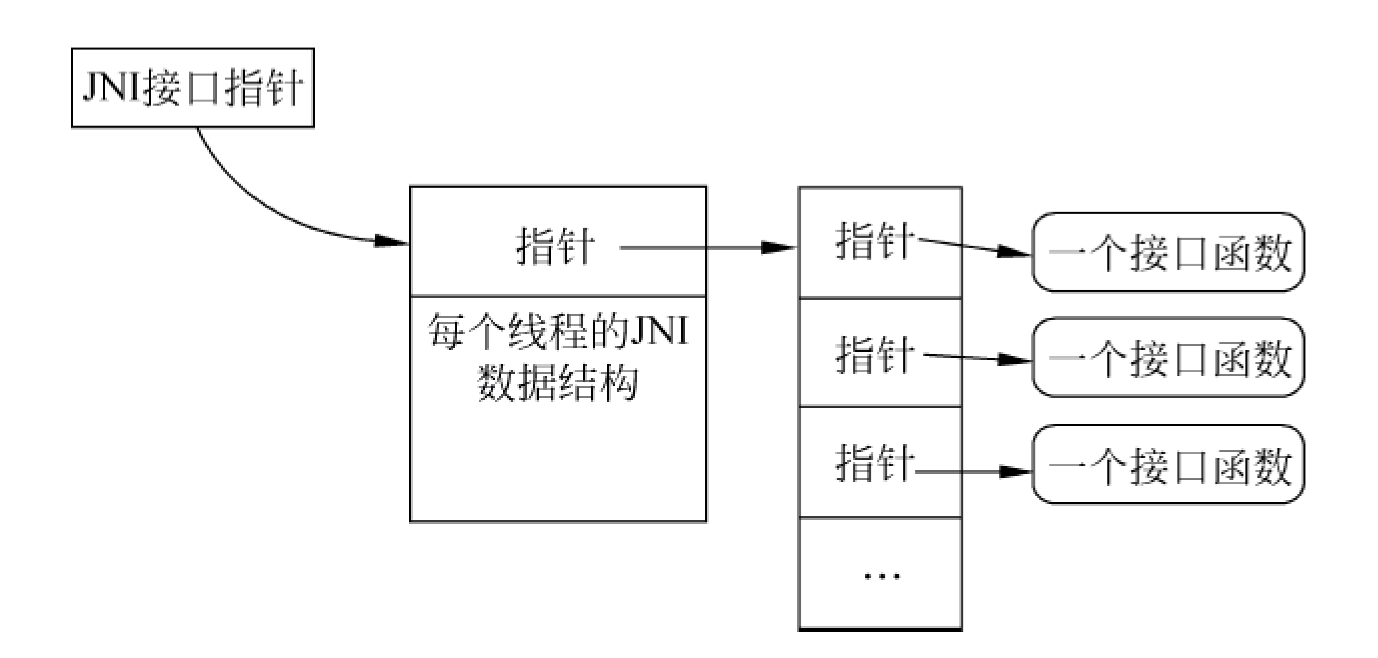


图 10-31　JNI 接口指针

Java 代码对本地库的加载通过 System.loadLibrary()方法实现。例如：

```
public native String stringFromJNICPP();

static {
    System.loadLibrary("TestEnv");
}
```

System.loadLibrary 的参数是编译生成的库的名称，这个参数不包含前后缀，系统按照标准的且与平台有关的处理方法根据这个参数转换得出本地库名，例如，Linux 系统将名称 TestEnv 转换为 libTestEnv.so，而 Win32 系统将相同的名称 TestEnv 转换为 TestEnv.dll。Java 代码初中通过 native 关键字声明的方法名将按照如下方法被解析为本地方法名从而找到该方法的具体实现，此处以 10.2.6 节的示例中的方法名为例：

- 前缀 Java_。
- 完整的包名并以下划线“_”代替“.”，即 com_android_example_。
- 类名 TestEnvActivity_。
- 方法名 stringFromJNICPP。

连接上述 4 步分别得到的字符串即可以得到该方法所对应的本地方法名为 Java_com_android_example_TestEnvActivity_stringFromJNICPP。该本地方法的实现为：

```
JNIEXPORT jstring JNICALL
Java_com_android_example_TestEnvActivity_stringFromJNICPP(JNIEnv * env, jobject obj)
{
    return env -> NewStringUTF("Hello From CPP");
}
```

其中，本地方法的第一个参数则是 JNI 接口指针，其类型为 JNIEnv。第二个 jobject 类型的参数即为 Java 对象的引用(即 TestEnvActivity 对象的引用)。由于该本地方法的 Java 声明中没有参数，因此此处只有两个参数，如果 Java 声明中有其他参数则应一次在后面列出。本地方法调用利用返回值将结果传回调用程序中。本地类型 jobject 即对应了 Java 类型的 Object，同样的还有 jclass，这些变量类型的原型、其他声明和 JNI API 都包含在 jni.h 头文件中，数据类型之间的对应关系如表 10-1 所示。

表 10-1 Java 类型与本地类型的对应关系

Java 类型	本地类型	说明
boolean	jboolean	无符号,8 位
byte	jbyte	无符号,8 位
char	jchar	无符号,16 位
short	jshort	无符号,16 位
int	jint	无符号,32 位
long	jlong	无符号,64 位
float	jfloat	32 位
double	jdouble	64 位
void	void	N/A

JNI 包含了很多对应于不同 Java 对象的引用类型，这些引用类型的组织层次如图 10-32 所示。

根据如图 10-32 所示的组织关系，每个 JNI 函数均可通过 JNIEnv 参数以固定偏移量进行访问。例如示例中使用如下代码来返回字符串：

```
return env -> NewStringUTF("Hello From CPP");
```

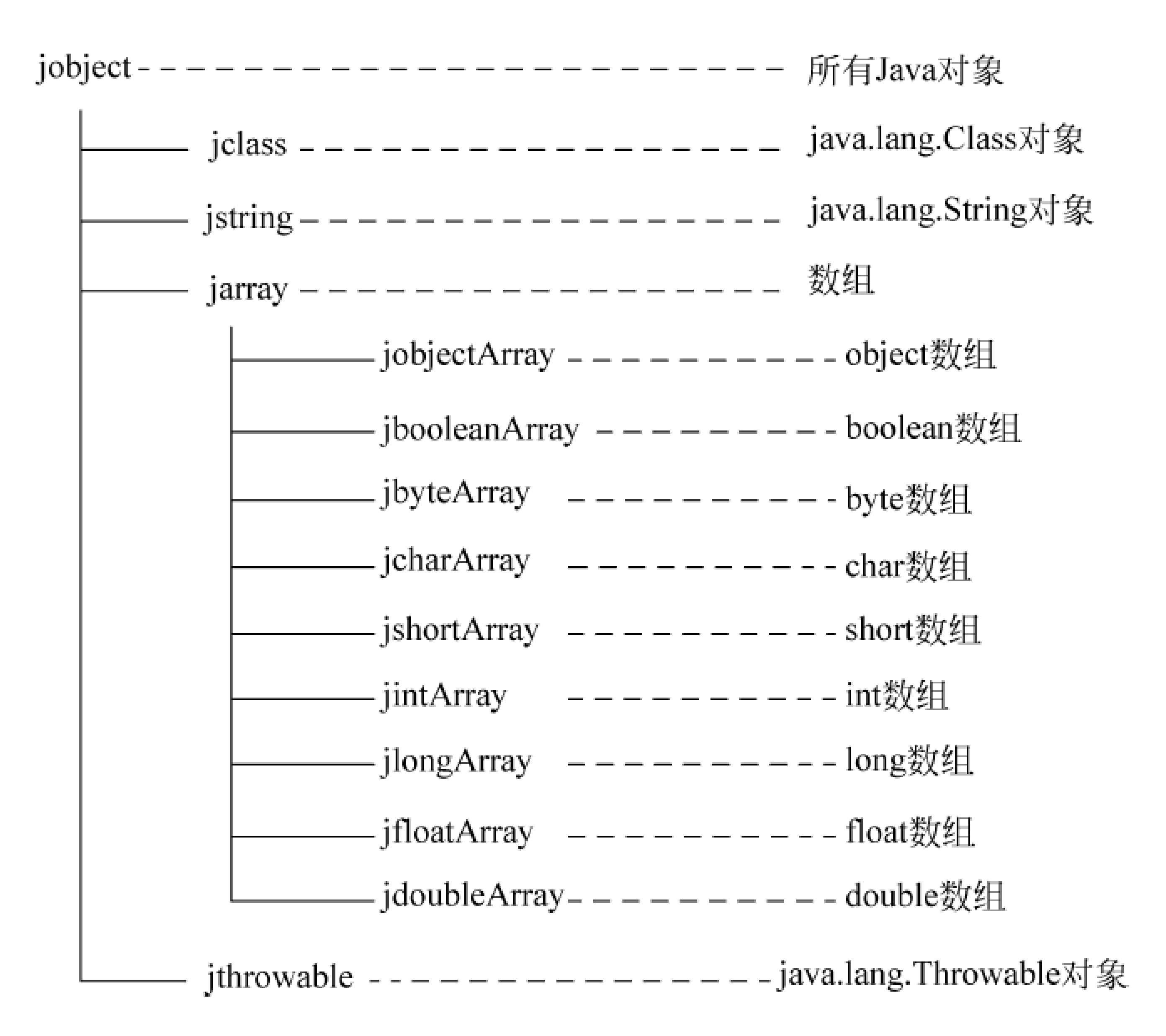


图 10-32　JNI 对 Java 对象的引用类型

下面简单列出一些常用到的 JNI 函数，需要注意的是，使用 C 和 C++ 调用时会有细微的差别。

（1）用于访问 jstring 对象的函数 GetStringUTFChars 和 ReleaseStringUTFChars：

```
const char * GetStringUTFChars(JNIEnv * env, jstring string, jboolean * isCopy);//C 形式
const char * str = ( * env) -> GetStringUTFChars(env, s, 0);                     //C 调用
const char * GetStringUTFChars(jstring string, jboolean * isCopy);               //C++形式
const char * str = env -> GetStringUTFChars(s,0);                                //C++调用
void ReleaseStringUTFChars(JNIEnv * env, jstring string, const char * utf);      //C 形式
( * env) -> ReleaseStringUTFChars(env,s,str);                                    //C 调用
void ReleaseStringUTFChars(jstring string, const char * utf);                    //C++形式
env -> ReleaseStringUTFChars(s,str);
```

（2）访问某个字段（field）的方法（C 形式）：

```
//获取 Java 对象
jclass GetObjectClass(JNIEnv * env, jobject obj);
//获取字段的 ID
jfieldID GetFieldID(JNIEnv * env, jclass clazz, const char * name, const char * sig);
//(一般性的表示方法,具体针对类型不同而不同)根据字段 ID 获取字段的值
NativeType Get<Type>Field(JNIEnv * env, jobject obj, jfieldID fieldID);
//通过字段 ID 来设定字段的值
void Set<Type>Field(JNIEnv * env, jobject obj, jfieldID fieldID, NativeType value);
```

（3）调用 Java 类中的某个方法（C++形式）：

```
//获取 Java 类型
jclass clazz = env -> GetObjectClass(obj);
//获取 Java 方法 ID,其中 method name 代表方法名,()V 为类型签名
```

```
jmethodID mid = env->GetMethodID(clazz, "method name", "()V");
//调用方法
env->CallVoidMethod(obj,mid);
```

还有许多其他方法例如数组操作，直接调用 Java API 的函数就不再一一列出，有兴趣的读者可以查阅 Java JNI 文档。对 JNI 的具体使用将在 10.3.2 节介绍。

10.3.2 NDK 示例

在 10.2.6 节中已经展示了一个 NDK 入门示例，示例通过调用本地方法获取了一段字符串并显示到 TextView 上，功能比较简单，本节将通过解析 NDK samples 中的 plasma 示例使读者加深对 NDK 使用方式的理解。NDK 所包含的 samples 是学习 NDK 的非常好的资料，这些 samples 也随着 NDK 的版本更新而更新，或者增添新的 sample，因此，对这些代码进行分析和理解对熟悉 NDK 有很大的帮助。目前 NDK 所带的 sample 主要包括如下几个：

- hello-jni——即在 10.3.1 节中用于验证开发环境的示例，这个示例的功能是从共享库中的一个 native 实现方法装载一个字符串，然后在程序的 UI 中显示出来。
- two-libs——顾名思义，这个示例涉及了两个库。本例中包含了一个静态库和一个共享库，而 native 方法是在静态库中实现的，共享库通过引入静态库的方式来使用 native 方法。
- san-angeles——使用 GLSurfaceView 对象管理 activity 的生命周期，并使用 native OpenGL ES API 着色 3D 图形。
- hello-gl2——使用 OpenGL ES 2.0 矢量和 fragment 阴影对一个三角形着色。
- hello-neon——显示如何使用 cpu feature 库来检查实时 CPU 兼容性，使用 NEON intrinsics。
- bitmap-plasma——演示在 native 代码中如何处理 Android Bitmap 对象的像素缓冲(pixel buffers)，然后产生经典的 plasma(等离子)效果。
- native-activity——演示如何使用 native-app-glue 静态库来实现 native activity。
- native-plasma——使用 native activity 的 bitmap-plasma 另一个版本。

1. 在 Eclipse 中建立 Plasma 项目

由于 NDK 所提供的示例并不是直接可使用的 Eclipse 项目，因此需要通过 New→Android Project→Create project from existing source 命令建立项目，如图 10-33 所示，单击 Finish 按钮即可得到新建立的 Plasma 项目，如图 10-34 所示。

同样，之后需要为该项目添加本机支持(Sequoyah 插件)，才能够同时进行本地库的开发。为了确保设置的库名正确，可以事先到 Plasma.java 代码里查看一下其 loadLibrary()方法的参数：

```
System.loadLibrary("plasma");
```

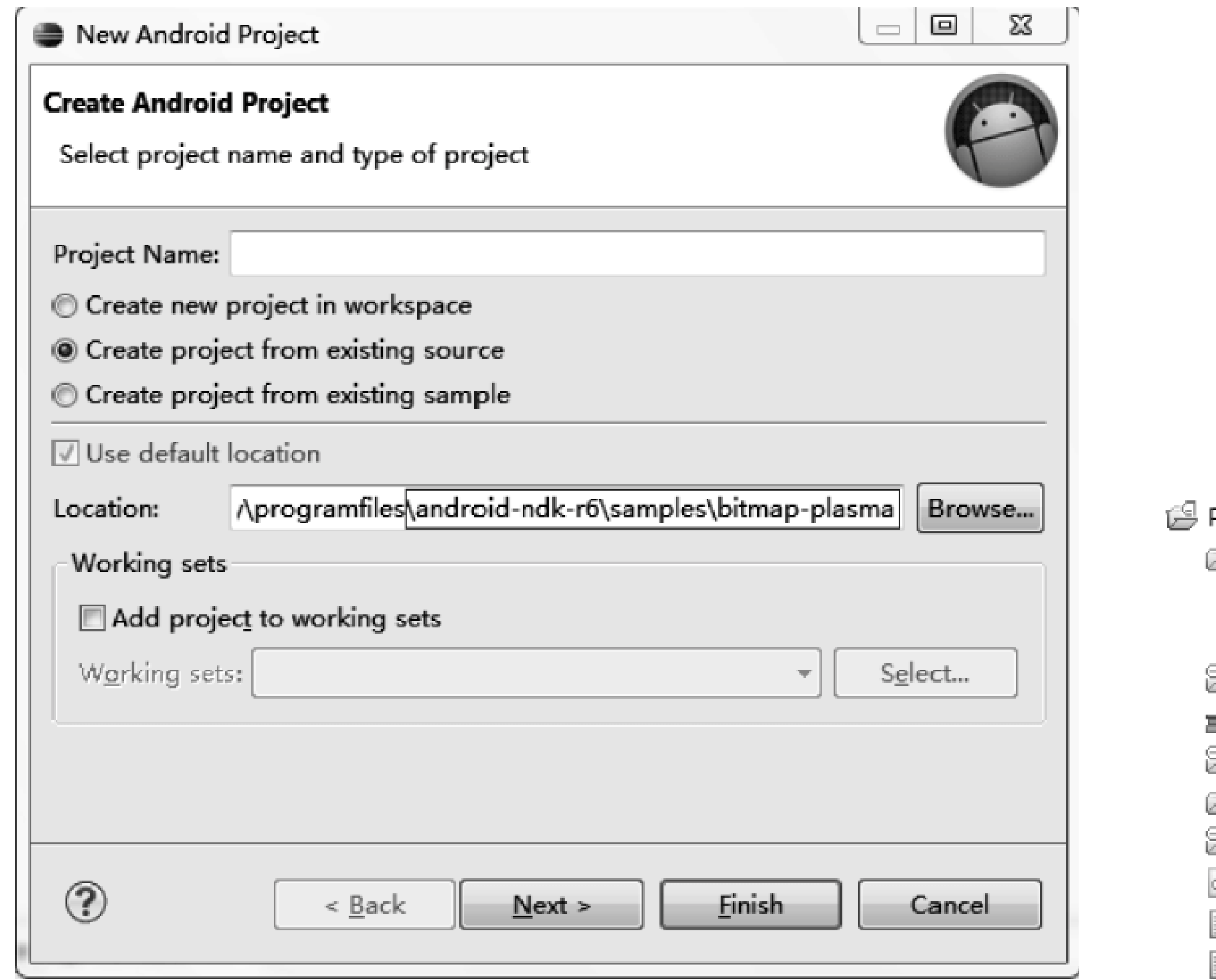


图 10-33　建立 Plasma 项目

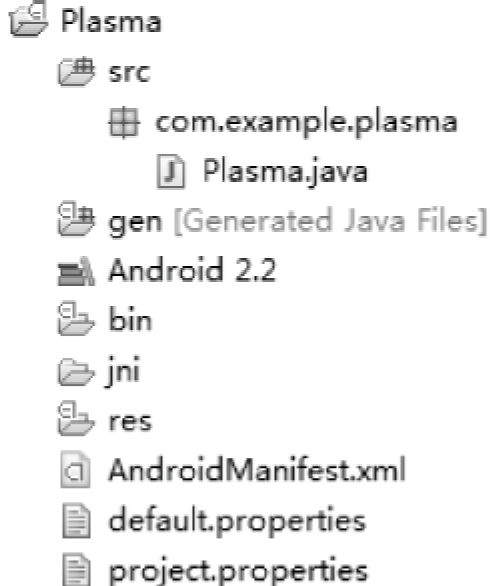


图 10-34　项目建立完成

可见，其库名称为 plasma，因此可在添加本机支持时填写库名称为 plasma，如图 10-35 所示，如 10.2.6 所提到的，确认 Eclipse 的 Project→Build Automatically 选项没有被选中，然后右击 Plasma 项目，在弹出的快捷菜单中选择 Build Project 命令，Build 成功后会生成相应的库，此时再切换到 C/C++视图下，可以看到项目的目录如图 10-36 所示。



图 10-35　添加 Android 本机支持

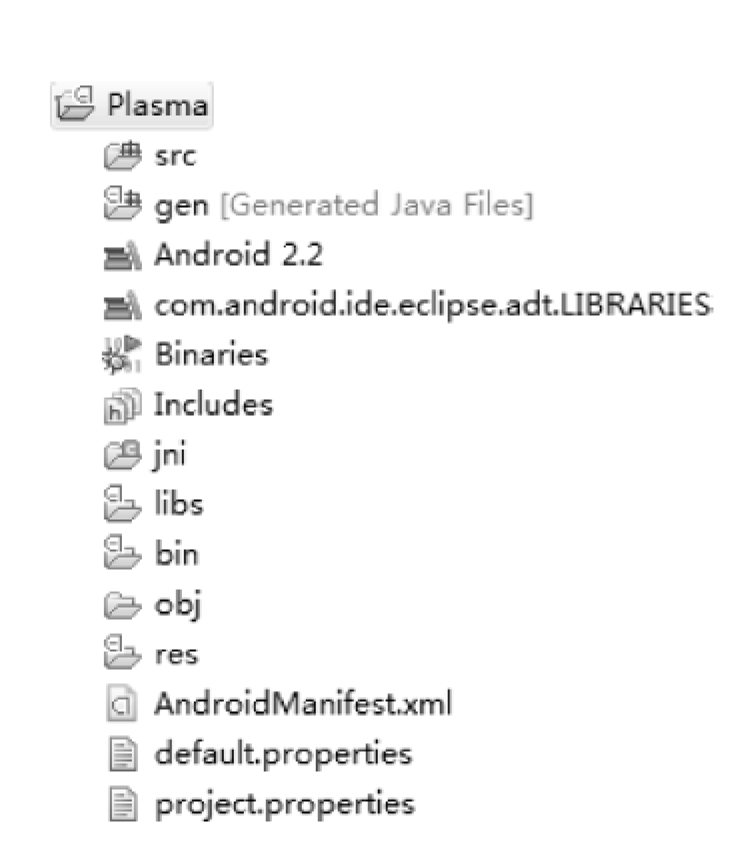


图 10-36　项目目录

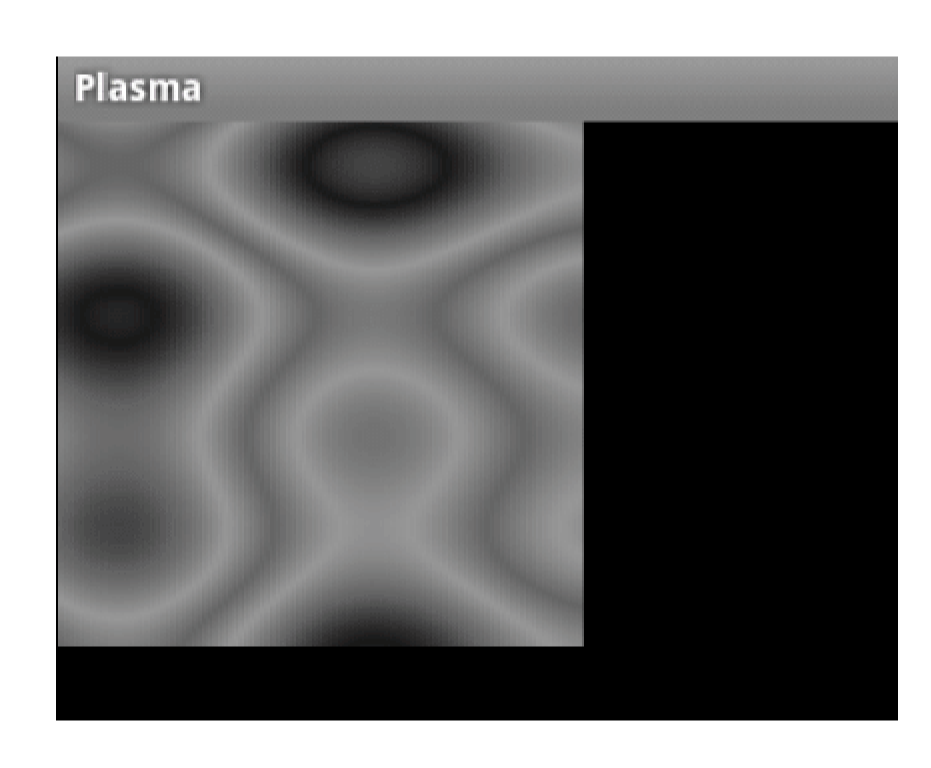


图 10-37 Plasma 运行效果

Build 完成后就可以在模拟器上运行该示例了，首先看一下该示例的运行效果，如图 10-37 所示。Plasma 是 2D 图像处理中一种经典的特效，它使用周期性变幻的色彩模拟出一种类似于液体流动的效果。

2. 代码分析

在示例效果中可以看到 Plasma 效果，形成这种效果的原理其实就是 bitmap 中的每一个像素点按照一点的规则使颜色连续变化。为什么能够形成这样的效果呢？本部分就从代码的角度来进行分析。首先来观察一下 Plasma.java 的代码，可以发现其与调用本地方法相关的代码有：

```
......
    /* load our native library */
    static {
        System.loadLibrary("plasma");
    }
......
class PlasmaView extends View {
    /* implementend by libplasma.so */
    private static native void renderPlasma(Bitmap bitmap, long time_ms);
......
    @Override protected void onDraw(Canvas canvas) {
        //canvas.drawColor(0xFFCCCCCC);
        renderPlasma(mBitmap, System.currentTimeMillis() - mStartTime);
        canvas.drawBitmap(mBitmap, 0, 0, null);
        // force a redraw, with a different time-based pattern.
        invalidate();
}
}
```

从代码中可以知道，在 Plasma 这个 Activity 启动时，会通过

```
System.loadLibrary("plasma");
```

来加载名为 plasma 的共享库，之后再在 PlasmaView 类中声明了名为 renderPlasma 的本地方法，该本地方法即是用于绘制 Plasma 特效的关键方法，可以看到该方法有两个参数：

- bitmap——即用于绘制的代表一帧的位图对象。
- time_ms——从参数名称可以猜测该参数代表了一个时间，结合代码的注释可以知道，这个参数利用了毫秒时间数值的递增性(结合三角函数便可以得到周期变化的数值，这就是 Plasma 特效的绘制原理)，使得 renderPlasma 方法能够根据此参数决定新的一帧的内容。

PlasmaView 通过 onDraw()方法来调用 renderPlasma()方法实现绘图，并通过在 onDraw()方法内部调用 invalidate()方法来强制重绘图像，从而得到一组连续变化的图像

效果。

通过分析 Plasma.java 的代码，知道了该项目的基本框架，接下来最主要的任务就是分析 renderPlasma()方法的具体实现过程。为此，进一步打开 jni 目录下的 plasma.c 文件查看代码。renderPlasma 方法的代码结构如图 10-38 所示。

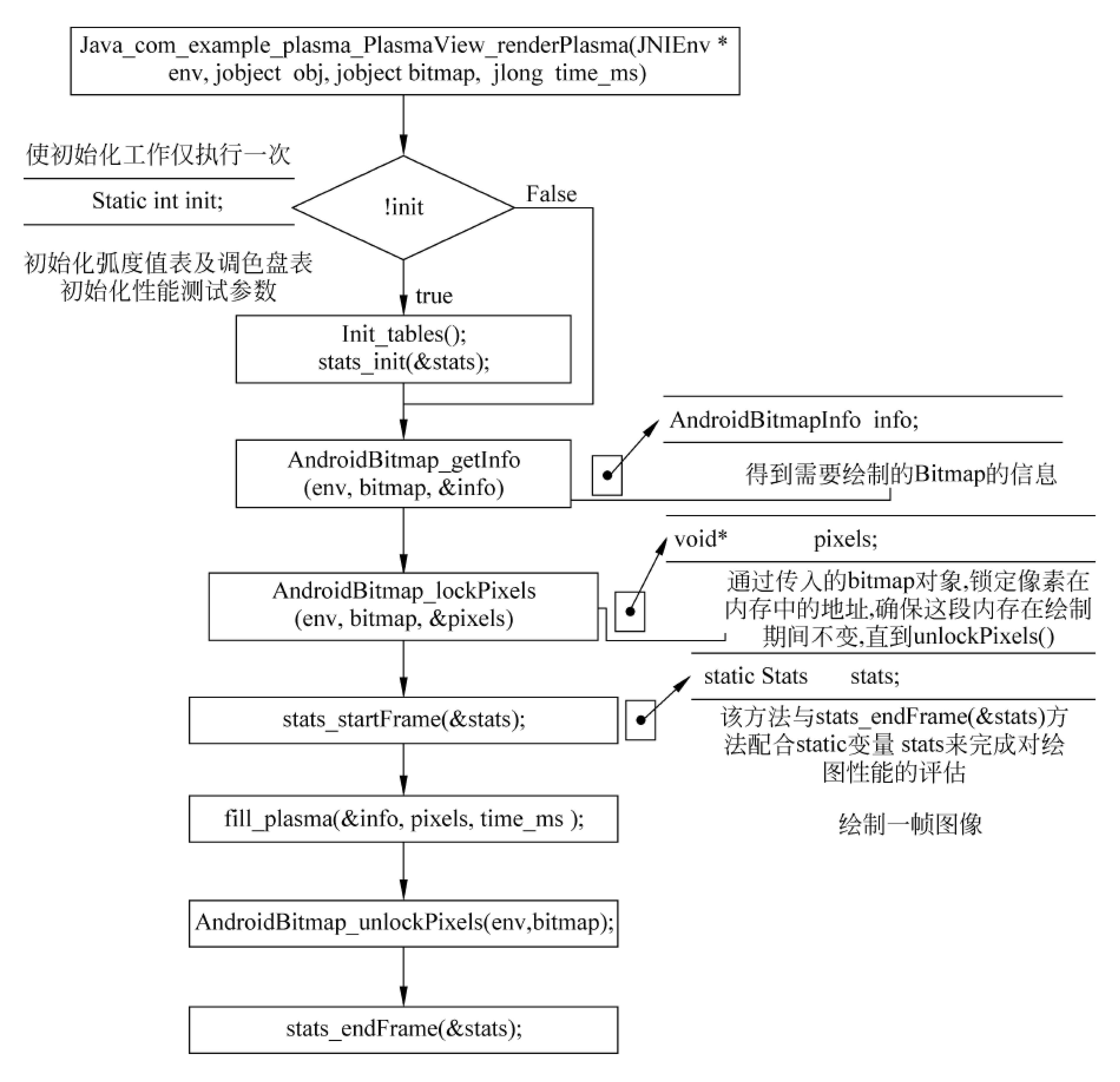


图 10-38 renderPlasma 方法的本地实现过程

如图 10-38 所示，renderPlasma()的本地方法主要经历了如下几个阶段：

- 初始化调色盘表—— init_palette()，这个方法将会得到一个数组 palette[PALETTE_SIZE]，通过查找这个数组的下标来得到对应代表的颜色，由 palette_from_fixed(Fixed x)方法完成查找。可以这样理解，即 bitmap 中的每一个像素都拥有一个数值，每个数值即代表了一种颜色。
- 初始化弧度值表——init_angles()，这个方法也将生成一个数组 angle_sin_tab[ANGLE_2PI+1]，对于一个 Fixed 类型的数值通过查找该数组来得到对应的弧度值。
- 测试相关——初始化测试所用参数 stats。
- 获取需要绘制的 bitmap 信息——AndroidBitmap_getInfo(env, bitmap, &info)，Android 提供的本地 API，该方法通过传入的 bitmap 得到其相关信息并存储在

info 中。

- 准备绘制 bitmap——AndroidBitmap_lockPixels(env, bitmap, &pixels)，本地API，该方法将尝试锁定像素在内存中的地址，确保这段内存在绘制期间不变，直到unlockPixels()方法被调用。
- 测试相关——获取测试相关数据并存入 stats，即记录帧开始的时刻。
- 绘制 bitmap——该方法将绘制一次 bitmap，通过一个传入的时间值为基础，配合三角函数，在 X 和 Y 轴上按照特定的增量来计算出每一个像素的值，从而绘制出 plasma 的效果。
- 完成 bitmap 绘制——AndroidBitmap_unlockPixels(env, bitmap)，释放对内存的锁定。
- 测试相关——记录一帧完成时的相关数据。

参考文献

1. Android NDK Dev Guide(随 Android NDK 一同获取)：http://developer.android.com/sdk/ndk/index.html.
2. Android 官方文档 What si the NDK?：http://developer.android.com/sdk/ndk/overview.html.
3. Android NDK Eclipse 集成：http://blog.csdn.net/id19870510/article/details/5903101.
4. Creating your first Android JNI/NDK Project in Windows Eclipse with Sequoyah：http://permadi.com/blog/2011/09/creating-your-first-android-jnindk-project-in-eclipse-with-sequoyah/.
5. Setting up Automatic NDK Builds in Eclipse：http://mobilepearls.com/labs/ndk-builder-in-eclipse/.
6. JNI Design Overview：http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/design.html.

第 11 章 游戏开发入门

11.1 游戏简介

11.1.1 游戏的定义

游戏，从广义上来说，包括了人类甚至是所有动物的许许多多的日常行为。游戏的历史非常悠久，很难确定它究竟是从何时开始的，而对于游戏的理论研究则是从近代才开始的，目前对于游戏存在着如下几种理论：

- 本能说——这是由德国诗人、剧作家席勒所提出的一种游戏理论。这种理论认为，“人类在生活中要受到精神与物质的双重束缚，在这些束缚中就失去了理想和自由。于是人们利用剩余的精神创造一个自由的世界，它就是游戏。这种创造活动，产生于人类的本能”。席勒还说：“只有当人充分是人的时候，他才游戏；只有当人游戏的时候，他才完全是人。”
- 剩余能量说——这是由英国哲学家赫伯特·斯宾塞提出的一种游戏理论，它是对席勒的本能说的进一步补充。这种理论认为，“人类在完成了维持和延续生命的主要任务之后，还有剩余的精力存在，这种剩余的精力的发泄，就是游戏。游戏本身并没有功利目的，游戏过程的本身就是游戏的目的”。
- 练习理论——德国生物学家谷鲁斯对剩余能量说和本能说又进行了进一步的修正。这种理论认为，游戏不是没有目的的活动，游戏并非与实际生活没有关联。游戏是为了将来面临生活的一种准备活动。例如，小猫抓线团是在练习抓老鼠，小女孩给布娃娃喂饭是在练习当母亲，男孩子玩打仗游戏是在练习战斗，等等。
- 宣泄理论——这种理论是由弗洛伊德提出的，他认为游戏是用于满足被压抑的欲望的一种替代行为。

无论是上述的哪一种有关游戏的理论，都表明了游戏对于人类的重要性，事实上也是如此，幼儿进行游戏不仅能够得到锻炼身体的作用，而且还能够在游戏中进行思考从而活跃思维，青少年、成年人等进行游戏则可以增进彼此的感情，还能够陶冶情操。

11.1.2 电子游戏

游戏分为很多种，包括人体游戏（例如剪刀石头布）、运动类游戏（例如乒乓球）、桌面游戏（例如三国杀）、棋牌类游戏（例如斗地主）、电子游戏等，显而易见，本书将要讨论的游戏是

属于电子游戏范畴。

1. 各种电子游戏平台介绍

所谓电子游戏(国外通常也称为视频游戏,即 Video Game),是指人类通过电子设备例如专用游戏机(例如任天堂推出的 Game Boy 和 Wii、Sony 推出的 PlayStation 系列游戏机、微软推出的 Xbox 360、街机等,如图 11-1 所示)、桌面计算机以及手机等进行的一种游戏方式。值得一提的是,电子游戏对于设备性能的高需求是推动各种电子设备不断发展的强大动力,很多设备的测评都是根据是否能够流畅地运行大型视频游戏来进行的。



图 11-1　各种游戏机

目前专用游戏机市场基本上被几家大型游戏机厂商占据着,这类市场由于受到专有平台的限制,对开发者来说门槛相对较高,通常只有大中型的游戏公司会参与这些游戏机设备上游戏的开发,这类游戏机凭借良好的操作体验以及"只为游戏"的特性而受到游戏玩家的青睐。而对于开发者来说,桌面计算机以及手机游戏市场则是一个更为广阔的平台,由于桌面计算机以及手机的普及率更高,因此拥有更加广大的用户群,再加上这两种平台上所提供给开发者的支持非常完善,使得游戏开发过程变得有章可循,特别是计算机作为一个功能较为全面的平台,随着硬件的不断更新,几乎能够模拟实现任何专用游戏机的功能,各种操控装置都能够方便地接入到计算机,因此还有人预测在未来桌面计算机将会完全取代专用游戏机,因为几乎所有的游戏都能够移植到 PC 平台,并且拥有一样的用户体验。对于手机来说,它又具备了便携性的优势。手机几乎是人们不会离身的一件物品,使得手机游戏市场成为了一个最大的市场,随着移动计算技术的发展,手机的硬件配置水平甚至已经超越了近几年前的 PC 主机配置,使得手机上的游戏也能够拥有华丽的 3D 特效和流畅的操作体验,特别是加上手机所具备的丰富的传感设备(重力感应、陀螺仪、多点触摸屏),让游戏的操作方式更加多样化。

2. 电子游戏的分类

电子游戏的数量庞大，按照不同的分类方式可以将这些游戏分为多种类型，一般来说可以按照游戏平台、游戏人数以及游戏的玩法来进行分类。本节第1部分介绍的就是一些游戏平台的种类，不过值得一提的是最新发展起来的网页游戏，这种游戏不依赖某个确定的硬件平台，而是依赖于浏览器；按照游戏人数来划分可以分为单人游戏、多人游戏以及大型多人在线游戏（网络游戏）；按照游戏玩法来划分就比较多样化了，常见的分类有动作、体育、角色扮演（RPG）、冒险、益智、射击、塔防（TD）、第一人称射击游戏（FPS）、格斗、棋牌、模拟、竞速、策略、即时战略（RTS）、大型多人在线角色扮演（MMORPG）、休闲等。

这些对电子游戏的分类通常是相互重合的，即往往一款游戏可以同时被归入多个分类，例如魔兽争霸可以同时被划分为角色扮演、即时战略、冒险类等，不同的游戏分类适合于不同的用户群，因此在开发一款游戏之前应该首先根据游戏要面向的用户群来选择合适的游戏类型。下面列举一些比较流行的游戏。

- 仙剑奇侠传5，角色扮演类，其界面如图11-2所示。
- 上古卷轴5，角色扮演类，其界面如图11-3所示。



图11-2　仙剑奇侠传5



图11-3　上古卷轴5

- 水果忍者，益智类，其界面如图11-4所示。
- 愤怒的小鸟，益智类，其界面如图11-5所示。



图11-4　水果忍者



图11-5　愤怒的小鸟

- 植物大战僵尸，益智类、塔防类，其界面如图 11-6 所示。
- 魔兽争霸 3，即时战略类、角色扮演类、塔防类（取决于具体地图），其界面如图 11-7 所示。



图 11-6 植物大战僵尸



图 11-7 魔兽争霸 3

- 战地 3，第一人称射击类，其界面如图 11-8 所示。
- 刺客信条：启示录，动作类、冒险类，其界面如图 11-9 所示。



图 11-8 战地 3



图 11-9 刺客信条：启示录

- FIFA12，体育类，其界面如图 11-10 所示。
- 真实赛车 2，竞速类，其界面如图 11-11 所示。



图 11-10 FIFA12



图 11-11 真实赛车 2

- 魔兽世界：大地的裂变，大型多人在线角色扮演类，其界面如图 11-12 所示。



图 11-12　魔兽世界：大地的裂变

11.2　Android 游戏开发入门

前面已经介绍了电子游戏平台类型以及电子游戏的类型，现在切入正题，开始介绍在 Android 平台上的游戏应用开发。在 11.1 节中已经知道了手机游戏市场的巨大，而在第 1 章中也已经介绍过，Android 已经成为了目前主流操作系统之一，因此游戏开发对于 Android 平台开发的重要性也就不言而喻了。

在互联网上有人做过一个统计，游戏是智能手机上最受欢迎的移动应用，64%的手机用户的手机上都安装有游戏，其次依次是天气预报（60%）、社交网络（56%）、地图和搜索（51%），仅仅有 32%的手机用户安装了银行或理财应用，只有 21%的用户安装了办公软件。虽然这对于提倡工作效率的今天来说并不是什么好的统计结果，但是这从另一方面反映了人们对于游戏应用的喜爱和需求，毕竟人不是机器，要是让一个人十年如一日地像机器一样工作，那么只会让他对工作越来越没有动力，越来越缺少激情。

在 Android Market 中也可以看到，游戏应用数量之大，甚至已经使得"游戏"作为一个独立的一级分类与"应用程序"并列，说不定到将来某一天，出现一个单独的"Android Game Market"也并非不可能。考虑到游戏开发在各种平台上具有很大的共性，因此本书通过例子来对 Android 游戏开发进行简要的介绍，相关游戏设计的一些基础技能和具体细节不再一一讲述，读者可以自行查阅专门的游戏开发类书籍。

11.2.1　Android 自带示例 Snake 简析

使用 Android SDK Manager 可以下载各个版本 SDK 的一些示例，目前可以下载到的示例中包括了 JetBoy（飞行射击）、LunarLander（月球登陆舱着陆）、Snake（贪吃蛇）和 TicTacToe（井字棋）这 4 个游戏示例，读者在进行游戏开发的过程中可以参考这几个游戏示例，其中 LunarLander 游戏还使用了传感器来进行控制。本节就来简单分析一下 Snake 游戏的代码实现。

1. Snake 示例效果

首先将该示例项目添加到 Eclipse 中，运行示例可以看到如图 11-13 所示的效果。

根据文字提示可以知道需要通过按键盘的上方向键来开始游戏，按方向键上开始游戏后如图 11-14 所示，其中蛇的头部为黄色块，身体为红色块，最外围的一圈绿色块代表着墙壁，在绿色围墙中会随机出现独立的代表食物的黄色块，游戏规则就是经典的贪吃蛇规则：蛇头以一定的速度向前移动，蛇身则以相同的速度跟随蛇头移动（蛇形前进），玩家可以使用上下左右键来控制蛇前行的方向，当蛇头碰到食物时，食物被蛇吃掉，玩家得分，蛇的身体增加一个单位（如图 11-15 所示），随着蛇吃下的食物增多，蛇头移动的速度也增快，通过增快移动速度这种方式来提高游戏难度，最后直到蛇头碰到墙壁或者蛇身则游戏结束（如图 11-16 所示，蛇碰到墙壁后已经消失）。



图 11-13　Snake 初始界面

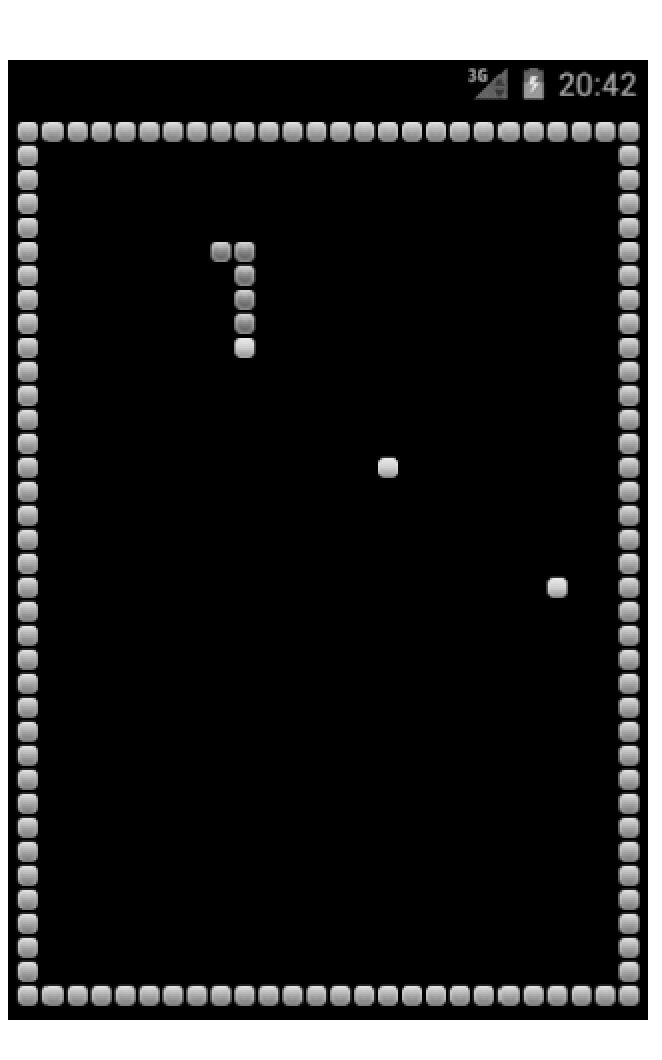


图 11-14　开始 Snake 游戏

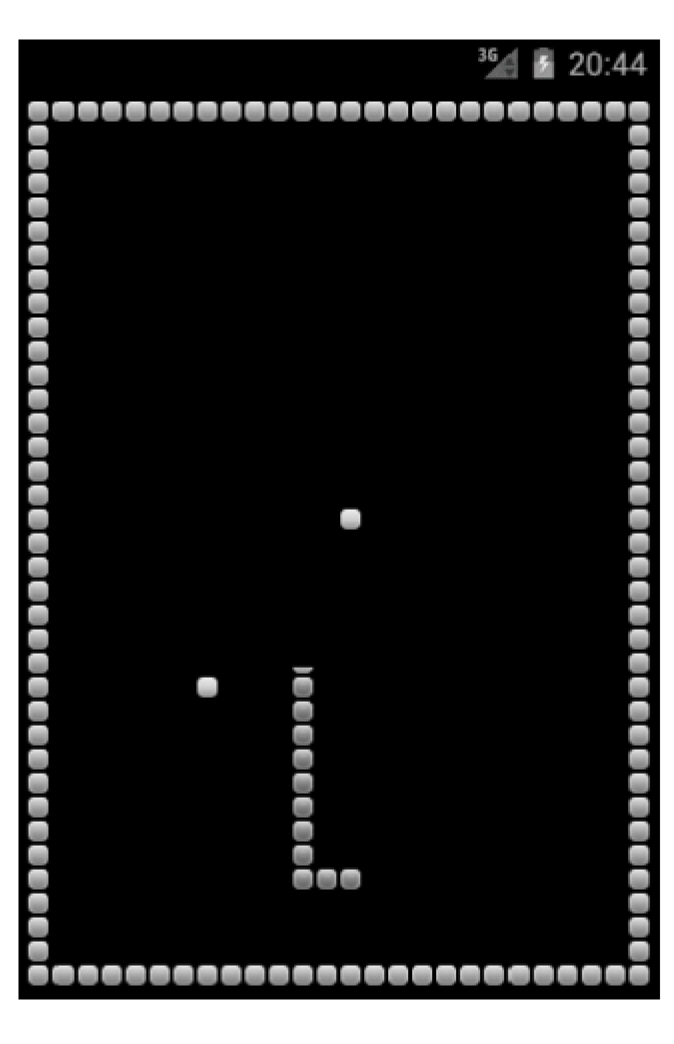


图 11-15　吃食物身体变长



图 11-16　碰到墙壁游戏结束

如上就是这个贪吃蛇项目所实现的功能，概括起来主要实现了如下几点：

- 游戏状态的切换，可以看到包括了等待开始、进行游戏、游戏结束这 3 个状态，实际

上该项目还实现了暂停状态(按 Home 键退出或者其他应用程序视图出现在屏幕最上方即进入暂停状态,按 Back 键会直接退出游戏而不进入暂停状态)。

- 对 TileView 的实现,即实现了一个简易的“模拟世界”,在这个模拟世界中,可以按需求“添砖加瓦”,例如修建墙壁、放置食物等。
- 对 Snake 游戏的驱动功能,即使得在正常游戏状态下蛇能够以一定的速率前进,从而推动游戏的进行。
- Snake 游戏的逻辑实现,即蛇吃食物、身体增长、移动速度增长、判断游戏结束等。

2. Snake 代码分析

查看项目源码可以看到该项目主要包含了 3 个类,即 Snake、SnakeView 和 TileView。其中 Snake 类是该游戏的 Activity,SnakeView 则负责实现贪吃蛇游戏的逻辑,SnakeView 继承了 TileView 类,TileView 类是负责绘制游戏视图的类,它又继承自 View,从类的名称可以知道,Tile 中文意为“瓷砖、瓦片”,TileView 作用正是将一个视图以块为单位进行分割,这是因为所要实现的贪吃蛇游戏所需要的就是这样可以以块为单位进行操作的视图,有了 TileView 作为基础,就可以在上面绘制代表墙壁的块、代表蛇身体的块以及代表食物的块了。

1) TileView

首先来分析一下 TileView 类的实现。TileView 是通过当前可用的屏幕大小来实时地对视图进行计算的,主要是确定出如下几个参数的值:

```
protected static int mTileSize;
protected static int mXTileCount;
protected static int mYTileCount;
private static int mXOffset;
 private static int mYOffset;
```

其中,最关键的一个值是 mTileSize,它代表了划分成块的尺寸,有了这个尺寸,再根据当前可用的屏幕大小可以计算出后面 4 个参数,它们分别代表了 X/Y 方向上块的数量以及 X/Y 方向上的初始位移(即整个可用区域的左上角坐标,如果屏幕的长宽能够被块的尺寸整除,那么这两个值也可能为 0)。这 4 个值的计算代码被包含在 onSizeChanged()方法内,这个方法将在当前 View 所占有的屏幕尺寸发生变化时被调用(也会在第一次显示时被调用),方法代码如下:

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    mXTileCount = (int) Math.floor(w / mTileSize);
    mYTileCount = (int) Math.floor(h / mTileSize);

    mXOffset = ((w - (mTileSize * mXTileCount)) / 2);
    mYOffset = ((h - (mTileSize * mYTileCount)) / 2);

    mTileGrid = new int[mXTileCount][mYTileCount];
    clearTiles();
}
```

在上面的代码中，第 03 行和第 04 行计算出了 X 和 Y 方向上可以存在的块的数目，由于可用屏幕的宽和高并不一定能够整除 mTileSize，为了能够居中显示游戏区域，在代码第 06 行和第 07 行计算了游戏区域的原点偏移量。通过这 4 行代码就确定了游戏区域的布局，将游戏区域分成了若干个（mXTileCount 乘以 mYTileCount）方块并且居于屏幕中央显示，具体的划分效果如图 11-17 所示，其中最上方线条代表了可用屏幕的边界，左上方线条交叉得到的就是最左上的方块位置，可以看到这个方块相对于可用屏幕的上方和左方都有一定的间隔存在，这两个间隔就是由第 06 行和第 07 行所计算出来的。根据这两个偏移量，再加上 mTileSize，就可以计算出任意具体方块的偏移量了。

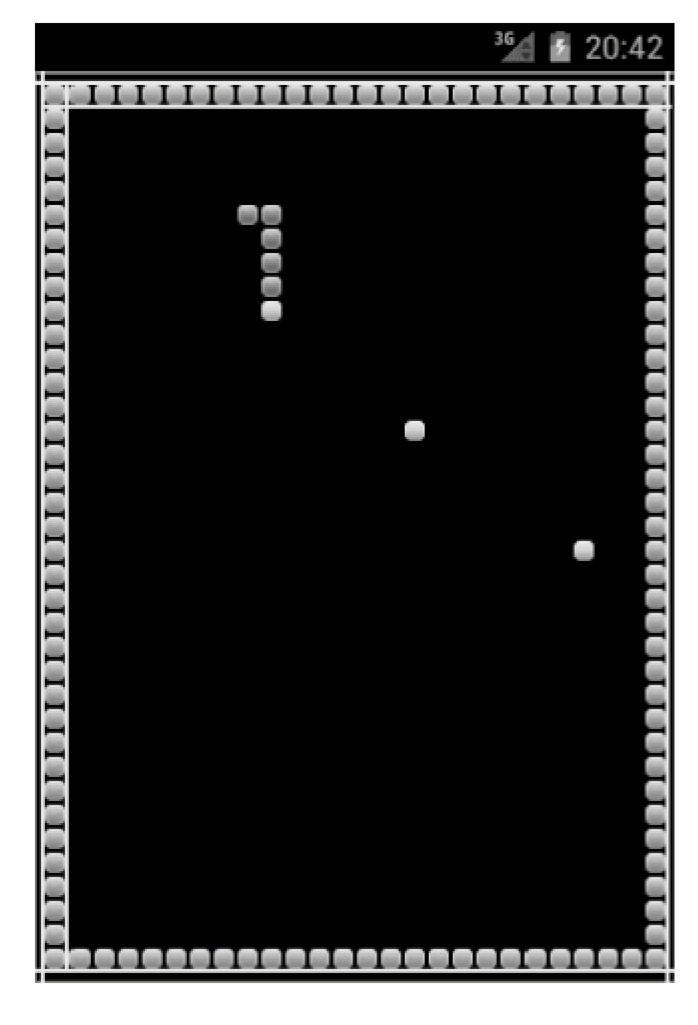


图 11-17　Snake 网格化示意图

第 09 行所涉及的二维整型数组 mTileGrid 存放了与前面所述的若干个方块一一对应的值，这些值代表了其所对应的块所应该绘制的图形，本例中一共有 3 张不同的图片，分别是绿色、红色和黄色的方块形图片，例如在图 11-17 中，mTileGrid[0][0] = 3 就确定了最左上角的块中绘制的是绿色方块形图片。这个数值与图片的对应关系是通过 TileView 的 loadTile()方法确定的，该方法的代码如下：

```
01 public void loadTile(int key, Drawable tile) {
02     Bitmap bitmap = Bitmap.createBitmap(mTileSize, mTileSize, Bitmap.Config.ARGB_8888);
03     Canvas canvas = new Canvas(bitmap);
04     tile.setBounds(0, 0, mTileSize, mTileSize);
05     tile.draw(canvas);
06
07     mTileArray[key] = bitmap;
08 }
```

该方法根据传入的参数来建立起数值与图片的联系，第 07 行涉及的一维数组（Bitmap 类型）mTileArray 就是用来保存这个数值与图片的对应关系的。SnakeView 的初始化方法 initSnakeView()调用了这个方法来实现了具体的建立联系操作：

```
private static final int RED_STAR = 1;
private static final int YELLOW_STAR = 2;
private static final int GREEN_STAR = 3;
private void initSnakeView() {
    setFocusable(true);

    Resources r = this.getContext().getResources();

    resetTiles(4);
    loadTile(RED_STAR, r.getDrawable(R.drawable.redstar));
    loadTile(YELLOW_STAR, r.getDrawable(R.drawable.yellowstar));
    loadTile(GREEN_STAR, r.getDrawable(R.drawable.greenstar));
```

```

}
```

上述代码建立了如下的关系映射：1 代表红色方格图片，2 代表黄色方格图片，3 代表绿色方格图片。如此一来，只需要维护 mTileGrid 这个二维数组即可，为此，TileView 提供了如下两个方法来操作这个数组：

```
public void clearTiles() {
    for (int x = 0; x < mXTileCount; x++) {
        for (int y = 0; y < mYTileCount; y++) {
            setTile(0, x, y);
        }
    }
}
public void setTile(int tileindex, int x, int y) {
    mTileGrid[x][y] = tileindex;
 }
```

其中，clearTiles()方法用于将数组清零，即相当于清空当前屏幕显示，而 setTile()方法则用于为指定坐标的方格赋值。

除了前面完成的一系列工作之外，还需要重写 View.onDraw()方法，才能够真正将 TileView 绘制出来：

```
@Override
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    for (int x = 0; x < mXTileCount; x += 1) {
        for (int y = 0; y < mYTileCount; y += 1) {
            if (mTileGrid[x][y] > 0) {
                canvas.drawBitmap(mTileArray[mTileGrid[x][y]],
                        mXOffset + x * mTileSize,
                        mYOffset + y * mTileSize,
                        mPaint);
            }
        }
    }
}
```

两个嵌套的 for 循环依次绘制每一个方块，每个方块要绘制的图片由 mTileArray[mTileGrid[x][y]]确定，每个方块的偏移量则由 mXOffset + x * mTileSize，mYOffset + y * mTileSize 来确定。

2）SnakeView

接下来分析一下 SnakeView 的代码，SnakeView 继承了 TileView，它将图像绘制工作交给了 TileView，自身主要实现的是游戏逻辑，SnakeView 包括了以下主要的一些方法：

- initSnakeView()——在 TileView 中已经介绍，主要是完成设置图片资源的工作。
- initNewGame()——用于初始化游戏，包括了清理掉原有的蛇、食物、创建新的蛇、添加初始的两个食物、重置行走速度和分数等操作。

- coorArrayListToArray()与 coordArrayToArrayList()——这两个方法互逆，实现的是 ArrayList 到 Array 的相互转换功能，这两个方法主要是为后面的保存和恢复游戏状态(saveState()和 restoreState())这两个方法服务的，由于在保存程序状态时通常使用 Bundle，而 Bundle 并没有提供对 ArrayList 类型对象的保存机制，因此需要将 ArrayList 对象转换为 Array 类型(可以理解为“编码”)，在需要恢复程序状态时再将 Array 解析为 ArrayList 即可(可以理解为“解码”)。
- saveState()与 restoreState()——分别用于保存游戏状态和恢复游戏状态。
- setTextView()——为 SnakeView 设置需要使用的 TextView(在 Activity 类 Snake 中调用)，这个 TextView 用于显示游戏相关信息(例如“Game Over”)。
- setMode()——设置游戏的状态。
- addRandomApple()——向地图中随机添加一个食物，包含了用于产生随机坐标点的算法和防止重叠的算法。
- update()——该方法周期性地被调用，方法内向下调用了分别用于更新墙壁、蛇和食物的方法 updateWalls()、updateSnake()和 updateApples()，同时调用 mRedrawHandler 的 sleep()方法，该对象将在随后介绍。
- updateWalls()、updateApples()与 updateSnake()——当 update()方法被调用时，即游戏向前推进时，计算并且更新下一个状态的墙壁、蛇和食物，其中 updateSnake()方法较为复杂，包含了对蛇进行整体移动、吃食物操作、死亡状态监测等逻辑。

SnakeView 除了包含了如上一些主要方法之外，还包含了两个重要的内部类。第一个是前面提到的 mRedrawHandler，它属于 RefreshHandler 类，这个类继承了 Handler，它能够接收并处理发送给它的消息(由 handleMessage()方法处理)，其代码如下：

```
class RefreshHandler extends Handler {

    @Override
    public void handleMessage(Message msg) {
        SnakeView.this.update();
        SnakeView.this.invalidate();
    }

    public void sleep(long delayMillis) {
        this.removeMessages(0);
        sendMessageDelayed(obtainMessage(0), delayMillis);
    }
};
```

游戏通常需要一个引擎来驱动其进行，最简单的游戏引擎就是 while()循环，但是使用 while()循环就需要另外开启一个线程，而线程的管理又是一个相对复杂的工作，因此示例中并没有使用 while()循环，而是巧妙地使用了 Handler 通过自身向自身发送消息来构成循环，这个循环的过程为：update()→mRedrawHandler.sleep()→sendMessageDelayed()→handleMessage()→update()。有了这个循环的过程，游戏的引擎就实现了，循环的间隔由 sendMessageDelayed()的参数确定，这个参数也确定了蛇的移动速度，通过减小这个参数的值就可以达到提升蛇的速度的效果。

另一个内部类 Coordinate 比较简单，它的作用就是封装一个坐标的信息。

SnakeView 通过重写 onKeyDown()方法来实现用户对游戏操作的接口，主要是实现了对上下左右按键事件的响应，从而使得用户可以使用上下左右键来控制蛇的方向，同时上键还额外承担了开始游戏和继续游戏的工作。

SnakeView 最关键的两个方法是 onKeyDown()和 updateSnake()，对这两个方法的理解需要结合贪吃蛇游戏的玩法来进行，读者在了解了贪吃蛇游戏的玩法之后，再来看这两个方法的代码，就会十分容易了。为了节省篇幅，这里就不对贪吃蛇游戏的逻辑实现代码进行分析了，因为在 11.2.2 节中将带领大家一步一步地去实现另一个经典的小游戏——俄罗斯方块，读者将能够看到整个逻辑代码的实现过程。

另外，该示例的 Activity-Snake 类比较简单，只包含了一些控件的绑定以及状态恢复代码，因此就不再对其进行分析。

11.2.2　俄罗斯方块的实现

本节将实现另一款经典的小游戏——俄罗斯方块。俄罗斯方块(英文名称：Tetris，俄文名称：Тетрис)是一款曾经风靡全球的游戏，包括电视游戏和掌上游戏机游戏。它由俄罗斯人阿列克谢·帕基特诺夫(Алексей Пажитнов，1956—)发明，据说 Тетрис 这个名字来源于希腊语 tetra，意思是“四”，而他最喜欢的运动是网球(tennis)，于是他把这两个词合二为一，命名为 Tetris。这款游戏的特点是：规则简单，容易上手，但是它的游戏过程却是变化无穷的，要熟练地掌握其中的操作与摆放技巧，难度却不低，这些特点使它成为了那个时代的人们所痴迷的游戏，它曾经造成的轰动与造成的经济价值可以说是游戏史上十分有标志性意义的一件大事。

1. 游戏规则

在开始实现俄罗斯方块游戏之前，首先需要了解的是它的游戏规则，虽然在近些年的发展过程中俄罗斯方块游戏出现了很多变种，但是最经典的还是原版的俄罗斯方块，这里介绍一下原版俄罗斯方块的游戏规则。

1) 游戏区域大小

标准大小为宽度 10 格，高度 20 格，方块将从场地上方以一定的速度下落。

2) 方块的种类

俄罗斯方块包含了 7 种类型的方块，每一个方块都由四个点构成，并且每个方块都由一个英文字母来表示(根据形状)：

- I 形方块，形状如图 11-18 所示，一共包含 2 种状态，一次最多可消除 4 层方块。
- J 形方块，形状如图 11-19 所示，一共包含 4 种状态，一次最多可消除 3 层方块。

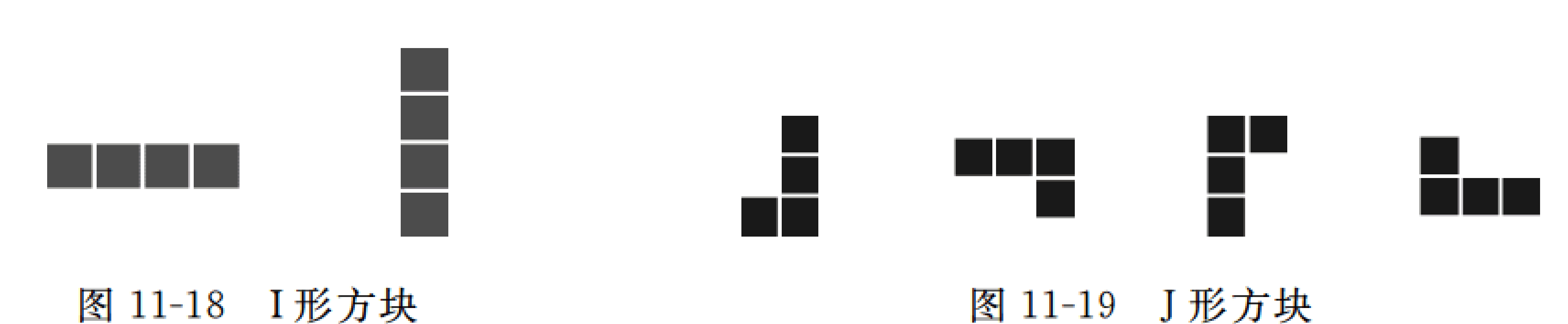


图 11-18　I 形方块　　　　图 11-19　J 形方块

- L形方块，它与J形方块互为镜像关系，形状如图11-20所示，同样包含了4种状态，一次最多可消除3层方块。
- O形方块，形状如图11-21所示，比较特殊的是它只存在一种状态，一次最多可消除两层方块。

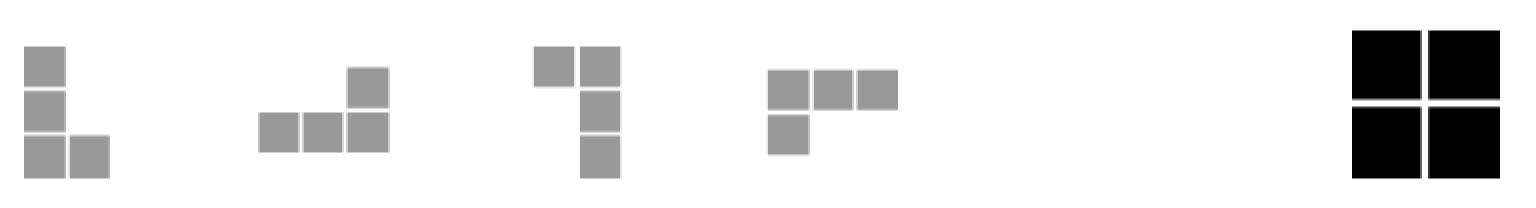


图11-20　L形方块　　图11-21　O形方块

- S形方块，形状如图11-22所示，它包含了2种状态，一次最多可消除2层方块。
- Z形方块，它与S形方块互为镜像关系，形状如图11-23所示，包括2种状态，一次最多可消除2层方块。在俄罗斯方块游戏中，S和Z形方块最容易造成空洞（一行有了空洞就使得这一行不容易被消除，导致方块累积并致使游戏结束）。



图11-22　S形方块　　图11-23　Z形方块

- T形方块，形状如图11-24所示，一共包含4种状态，一次最多可消除2层方块。

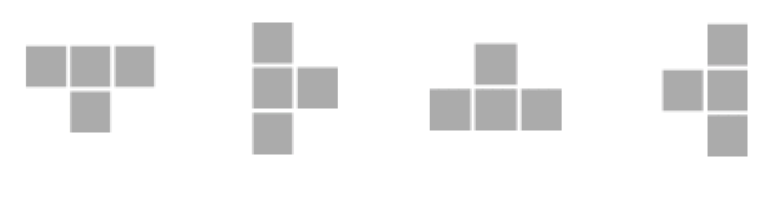


图11-24　T形方块

3）方块的变换和活动周期

游戏开始后，场地的上方会出现第一个随机的方块，在方块触地之前，这个方块就处于活动状态，它是可以被玩家所控制的，可以通过左右键左移或右移方块，通过上键变换方块的状态，使用下键加速方块下落等，一旦方块触地，那么该方块就会变为非活动状态，变成“地”的一员，场地上方会出现下一个活动方块。

4）方块预览

游戏时可以预览下一个即将到来的方块，玩家可以根据这一信息来决定如何放置当前处于活动状态的方块。

5）方块消除规则

当某一行的10格全部被方块填满时，这一行将被消除，上方的方块保持原样下落。

6）计分规则

当一行方块被消除时，玩家会得分，并且一次性消除的行数越多，玩家的得分也越多，例如，可以将计分指数设定为1、3、6、10。

7）游戏等级规则

为了提升游戏的难度，方块的掉落速度将会随着分数的增长而增加，这是因为如果掉落速度一直较慢，使得游戏能够不断进行下去，这对于商业用游戏是不理想的，为了加速游戏的结束，同时增加挑战性，加入了游戏等级规则。

8）游戏结束规则

当堆积的方块达到一定的高度时，游戏结束。

以上就是俄罗斯方块游戏的一些基本规则，下面的代码实现过程就是以这个规则为导

向的，在了解了这些游戏规则之后，就可以开始编写具体的代码了，本章配套的代码包含了一系列前缀为 Android2DTetris 的 eclipse 项目，根据后缀编号从小到大依次对应了实现过程中的各个阶段，读者可以参考这些代码来进行理解，观察这个游戏从开始到最后是怎样一步一步实现的。

2. 游戏区域的构造

首先需要实现的是游戏区域的构造，毕竟没有这个场地，方块就没有展现和放置的平台，就好比想修建房屋却没有陆地，恐怕就只有修建一座想象中的空中楼阁了。这时候前面分析过的 Snake 示例就能够派上用场了，因为从某种意义上来说，贪吃蛇和俄罗斯方块是一个类型的游戏，因为它们都是由一个一个的点构成的，不同的只是点在这两个游戏中代表的意义。但是这里并不打算完全照搬 Snake 的实现方式，因为 Snake 是基于 View 类的，而事实上在游戏开发中，通常会使用带有双缓冲机制的 SurfaceView，因此这里使用了 SurfaceView 来实现。在第 7 章讲解双缓冲技术时已经对 SurfaceView 的用法进行了说明，这里来看一看关键代码，首先是需要定义的一些常量和变量：

```
public class TetrisView extends SurfaceView implements SurfaceHolder.Callback {

    //当前占有屏幕的高度和宽度
    private int mHeightOfTheView;
    private int mWidthOfTheView;

    //游戏主区域的左上角位置
    private static int mXOffset;
    private static int mYOffset;

    //游戏主区域的高度和宽度
    private final int NUM_COLUMNS = 10;
    private final int NUM_ROWS = 20;

    private static int mTileSize;//每一格的尺寸
    private SurfaceHolder mSurfaceHolder;
    private Canvas mCanvas;
    private Paint mPaint;
……
```

如上面的代码所示，游戏区域尺寸和坐标的确定借鉴了 Snake 的方法（图 11-17 所示），不同的是 Snake 的确定性参数是 mTileSize，贪吃蛇的代码通过 mTileSize 来计算出 mXTileCount 和 mYTileCount，而俄罗斯方块的确定性参数则是 NUM_COLUMNS（mXTileCount）和 NUM_ROWS（mYTileCount），因为前面的游戏规则已经确定这个游戏区域的宽度为 10 个单位，高度为 20 个单位，根据这两个参数可以计算出 mTileSize 的最恰当值（最大值），然后又可以计算出游戏区域相对于屏幕的偏移量（偏移量的作用是使得游戏区域在屏幕中居中显示），从而构造出一个基本的游戏区域，这个游戏区域的每一个单元格都是相互独立的，具体计算的代码包含在 SurfaceView 的 surfaceCreated()方法内，即每当 SurfaceView 被显示的时候进行计算，这样做可以使得游戏能够适应屏幕的实时变化，代码如下：

```
01 @Override
02 public void surfaceCreated(SurfaceHolder holder) {
03     mHeightOfTheView = this.getHeight();
04     mWidthOfTheView = this.getWidth();
05 
06     //得到每个方块的尺寸
07     int tileMaxHeight = (int) Math.floor(mHeightOfTheView / NUM_ROWS);
08     int tileMaxWidth = (int) Math.floor(mWidthOfTheView / NUM_COLUMNS);
09     mTileSize = tileMaxHeight > tileMaxWidth ? tileMaxWidth : tileMaxHeight;
10 
11     //用于居中显示视图
12      mXOffset = ((mWidthOfTheView - (mTileSize * NUM_COLUMNS)) / 2);
13      mYOffset = ((mHeightOfTheView - (mTileSize * NUM_ROWS)) / 2);
14 
15      drawTheTiles();
16 }
```

在第 03～13 行计算出各个参数的值后，再在第 15 行调用了用于绘制图像的方法 drawTheTiles()：

```
private void drawTheTiles(){
    mCanvas = mSurfaceHolder.lockCanvas();
    mPaint.setColor(Color.WHITE);

    //绘制游戏场地
    mCanvas.drawRect(mXOffset, mYOffset, mXOffset + NUM_COLUMNS * mTileSize,
            mYOffset + NUM_ROWS * mTileSize, mPaint);
     mSurfaceHolder.unlockCanvasAndPost(mCanvas);
}
```

这里实现的是绘制出一片合适的游戏区域的功能，实际上就是绘制出一个矩形，上面的代码绘制出了一个白色的矩形，这个白色矩形就代表了俄罗斯方块的游戏区域，运行效果如图 11-25 所示。

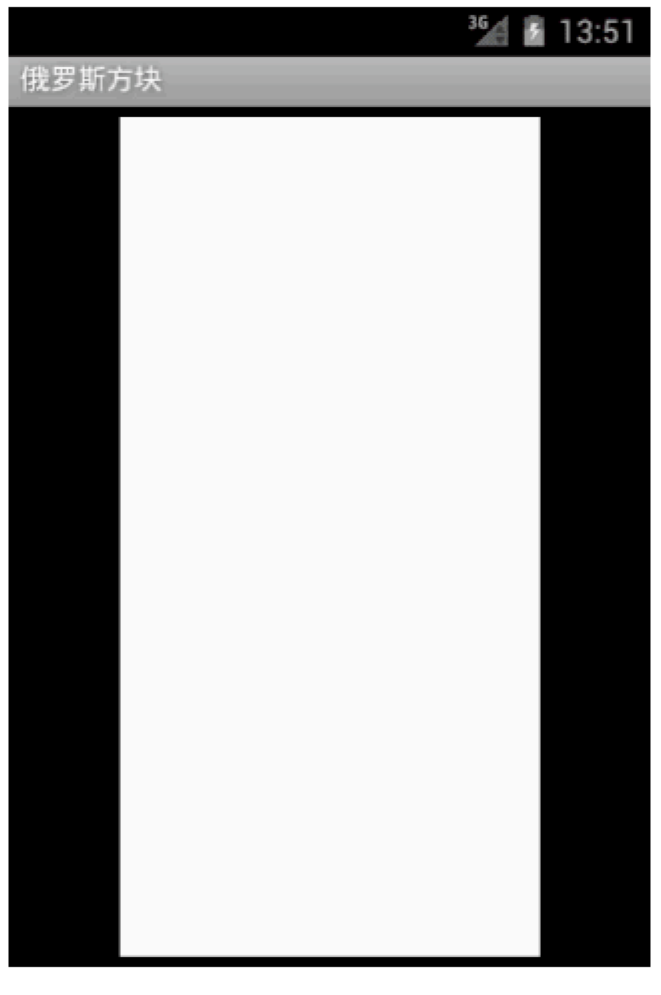


图 11-25 绘制出的游戏区域

3. 单元方格的绘制

构造好了游戏区域，就好比已经为房屋的建设买好了地盘，接下来需要的就是修建房屋的材料了，而在俄罗斯方块游戏里的材料，就是单元方格的确定。每个单元格所占有的区域与前面计算所得到的一系列参数相关，例如最左上角的一个单元格(0,0)所占有的区域为(屏幕左上角为原点，水平向右为 X 轴正方向，竖直向下为 Y 轴正方向)：

- X 轴方向为(mXOffset，mXOffset ＋ mTileSize)。
- Y 轴方向为(mYOffset，mYOffset ＋ mTileSize)。

对于游戏区域任一坐标(row，column)的单元格，很容易得到其占有的区域为：

- X 轴方向为(mXOffset ＋ column ＊ mTileSize，

mXOffset ＋（column ＋ 1） ＊ mTileSize）。

- Y 轴方向为（mYOffset ＋ row ＊ mTileSize，mXOffset ＋（row ＋ 1） ＊ mTileSize）。

得到了这样的计算公式后，就能够在 drawTheTiles()方法中独立地绘制每一个单元格了，接下来就利用这个关系在游戏区域中随意地绘制一些单元格，来测试绘制的效果。首先，为 TetrisView 增加一个常量：

```
private final int INTERVAL_BETWEEN_TILES = 1;
```

这个常量代表的是方格之间的间隔（1 个像素单位），这是为了能够清晰地分隔开每一个单元格，也同时能够实现在前面如图 11-18～图 11-24 所示的那种方块效果。引入这个间隔常量之后，每个单元格需要绘制的区域就变成了：

- X 轴方向为（mXOffset ＋ column ＊ mTileSize ＋ INTERVAL_BETWEEN_TILES，mXOffset ＋（column ＋ 1） ＊ mTileSize － INTERVAL_BETWEEN_TILES）。
- Y 轴方向为（mYOffset ＋ row ＊ mTileSize ＋ INTERVAL_BETWEEN_TILES，mXOffset ＋（row ＋ 1） ＊ mTileSize － INTERVAL_BETWEEN_TILES）。

然后需要定义一个二维数组，用于保存每一个单元格的信息：

```
//该数组代表了游戏主区域当前所有的方块
private int[][] mTileArray = new int[NUM_ROWS][NUM_COLUMNS];
```

可以使用类似于贪吃蛇程序的那种方式来实现，即在数组中存放整数，然后将这些整数映射成不同的图片资源，这里采取一种简便的方式，将方格绘制成纯色的，即直接在数组中存放方块的颜色值（RGBA 值），在绘制具体的方格时取出这个颜色值进行绘制即可，为了方便对这个二维数组的操作，添加如下两个方法：

```
//清除所有方块
private void clearTiles() {
    for (int row = 0; row < NUM_ROWS; row++) {
        for (int column = 0; column < NUM_COLUMNS; column++) {
            setTile(Color.WHITE, row, column);
        }
    }
}

//设置某个方块的颜色
private void setTile(int tileColor, int x, int y) {
    mTileArray[x][y] = tileColor;
}
```

这两个方法与贪吃蛇代码中的两个方法 clearTiles()和 setTile()的作用是一致的，唯一的不同是将 tileIndex 换成了 tileColor。根据前面的思路，在 drawTheTiles()方法中加入用于绘制方块的如下代码（代码添加到 mSurfaceHolder.unlockCanvasAndPost()方法之前）：

```
//绘制方块
for (int row = 0; row < NUM_ROWS; row++) {
```

```
        for (int column = 0; column < NUM_COLUMNS; column++) {
            if (mTileArray[row][column] != Color.WHITE) {
                mCanvas.save();
                mCanvas.clipRect(mXOffset + column * mTileSize + INTERVAL_BETWEEN_TILES,
                        mYOffset + row * mTileSize + INTERVAL_BETWEEN_TILES,
                        mXOffset + (column + 1) * mTileSize - INTERVAL_BETWEEN_TILES,
                        mYOffset + (row + 1) * mTileSize - INTERVAL_BETWEEN_TILES);
                mCanvas.drawColor(mTileArray[row][column]);
              mCanvas.restore();
            }
        }
}
```

至此，单元方格的绘制框架就已经完成了，要在游戏区域显示某些方块，只需要使用setTile()方法来修改mTileArray就可以了。为此，在surfaceCreated()方法中调用drawTheTiles()方法之前加入如下的测试代码：

```
clearTiles();
setTile(Color.BLACK, 5, 5);
setTile(Color.BLACK, 6, 5);
setTile(Color.BLACK, 5, 6);
setTile(Color.BLACK, 6, 6);
drawTheTiles();
```

要运行示例，为其添加一个TetrisActivity，该Activity仅仅需要将TetrisView设置为其主视图即可：

```
public class TetrisActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new TetrisView(this));
    }
}
```

然后运行示例，可以得到如图11-26所示的效果，可以看到游戏区域中出现了一个O形的方块。

同样，还可以绘制出其他种类的方块，只需要简单地使用setTile()方法设置单元格颜色即可，效果如图11-27所示。

4. 将方块封装为类

前面已经绘制出了所需的7种不同的方块，然而这只是表面上实现了而已，因为仅仅是绘制出了一些独立的点，如果通过直接对这些点进行操作来实现方块的移动和变化，代码会变得十分繁杂，因此这里考虑将每一种方块用一个类来进行封装，首先，定义了一个名为TetrisObject的接口，所有代表俄罗斯方块的类都必须实现这个接口，根据实际需要，该接口一共包含了3个方法，代码如下：

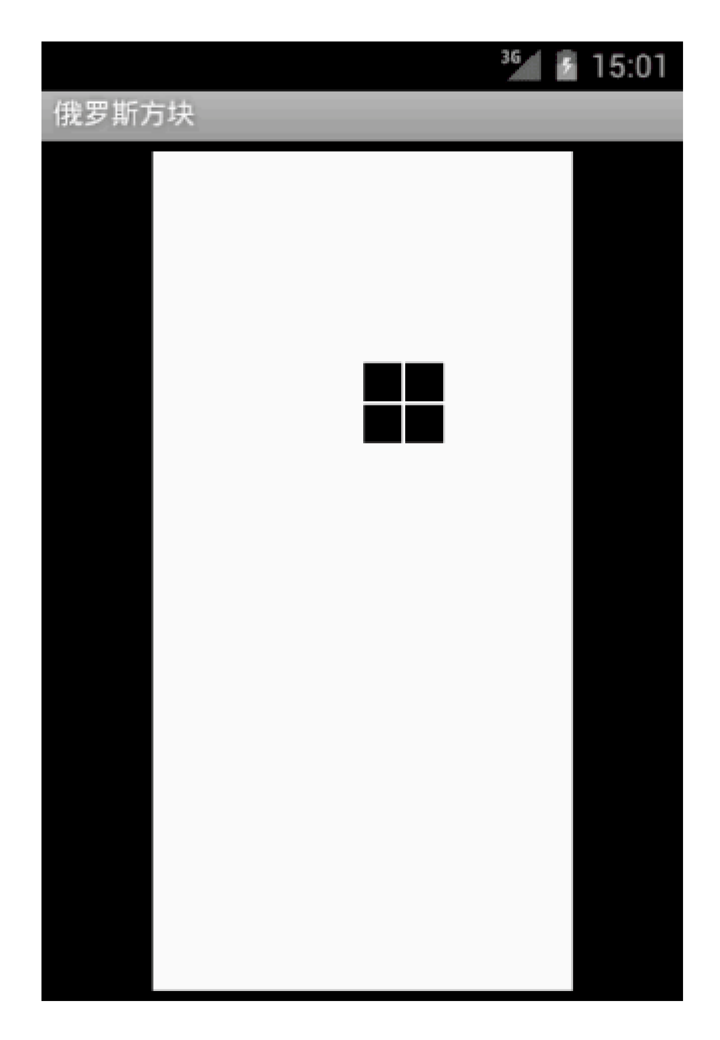


图 11-26　绘制一个 O 形方块

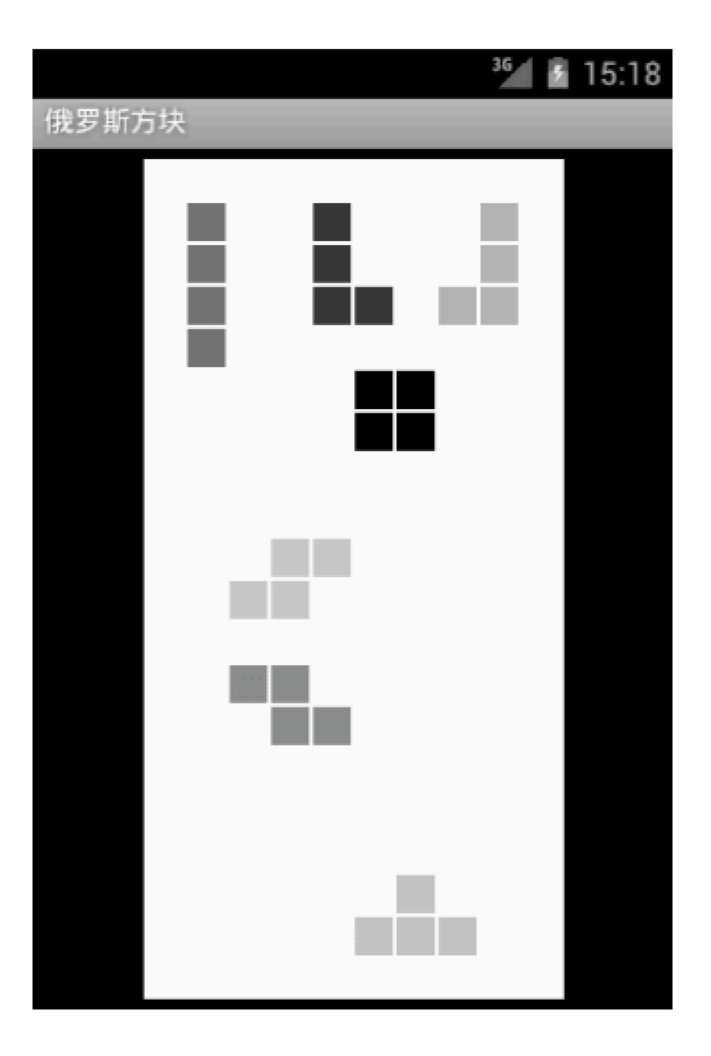


图 11-27　绘制各种形状的方块

```
public interface TetrisObject {

    public abstract void transform();

    public abstract Coordinate[] getTetrisForm();

    public abstract int getTetrisColor();

}
```

其中，transform()方法的作用是让当前的方块切换到下一个状态；getTetrisForm()方法的作用是得到当前方块所处的状态，方块的状态由一个 Coordinate 类型的数组表示，Coordinate 类也是来自于 Snake；getTetrisColor()方法的作用则是获取当前方块的颜色，有了 Coordinate[]数组和颜色值后，就能够绘制出指定状态的方块了。

定义好了 TetrisObject 接口之后，下面通过实现一个代表 O 形方块的类来快速验证我们的想法，O.java 的代码如下：

```
public class O implements TetrisObject {

    private final Coordinate[] stateOne = {new Coordinate(0, 0), new Coordinate(0, 1),
        new Coordinate(1, 0), new Coordinate(1, 1)};
    public Coordinate[] currentState = stateOne;

    public O(){

    }

    @Override
    public void transform() {
        currentState = stateOne;
```

```
14        }
15
16        @Override
17        public Coordinate[] getTetrisForm() {
18            return stateOne;
19        }
20
21        @Override
22        public int getTetrisColor() {
23            return Color.BLACK;
24        }
25
26    }
```

O类的实现非常简单，这是由于它仅仅存在一个状态，因此它的 transform() 和 getTetrisForm()方法仅仅只有一行代码，第 03 行和第 04 行定义了 O 形方块唯一存在的一种状态，需要注意的是方块状态的定义方式，采用的是方块所包含单元格的相对坐标，在接下来的实现中将会看到，当方块被构造并且添加到游戏区域中时，会被分配一个基准点坐标(basePoint)，方块在区域内的移动都是以这个基准点坐标为基础的，当方块的位置发生变化时，基准点的坐标会发生相同的变化，通过基准点坐标再加上方块所包含的单元格的相对坐标即可计算出方块实际的坐标，如图 11-28 所示。

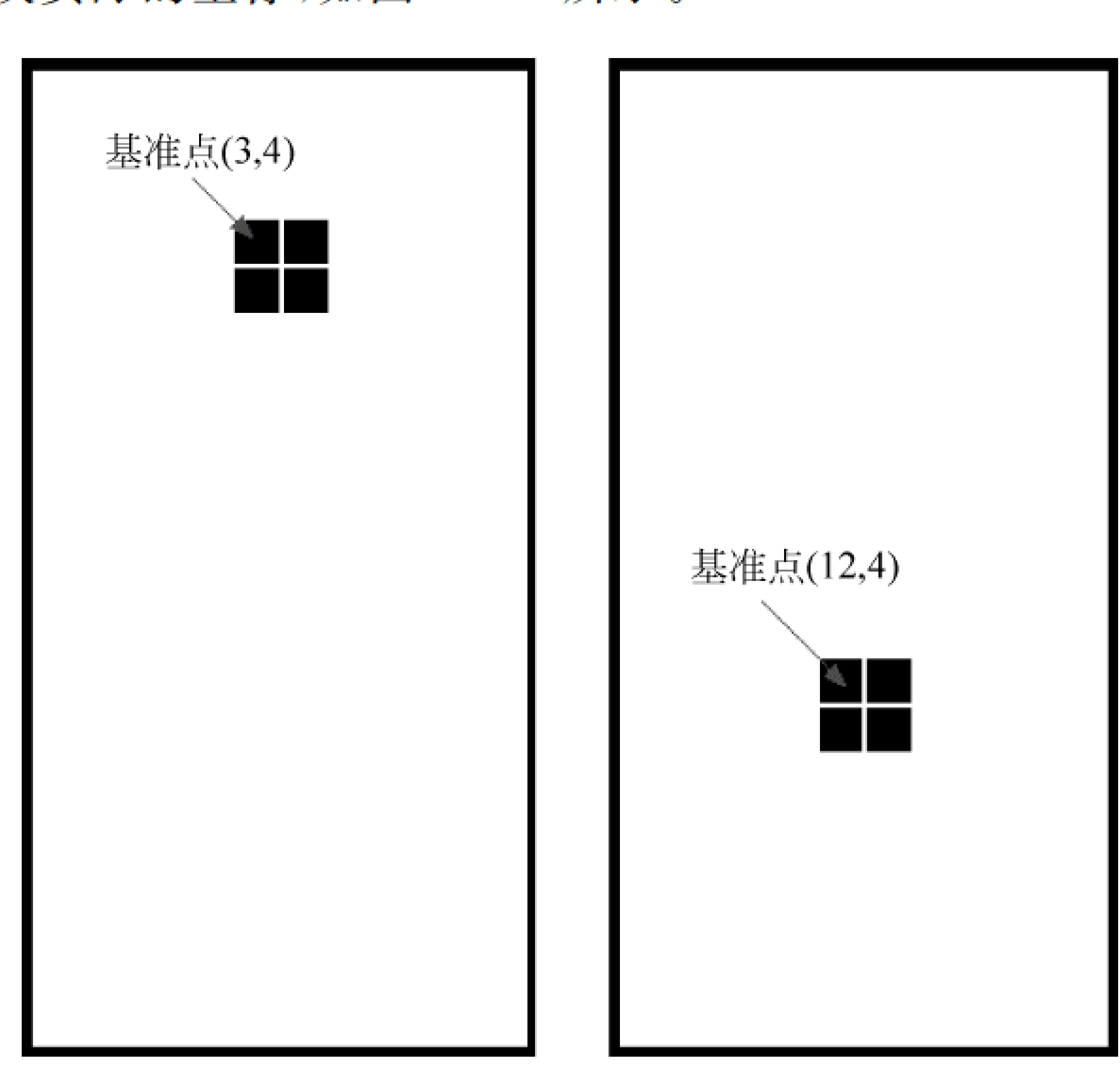


图 11-28　基准点与方块的关系

单单实现了 O 形方块还不足以说明其他方块类的实现模式，下面来实现具有 4 种状态的 L 形方块。首先，需要弄清楚 4 种状态下的 L 形方块所包含单元格的相对坐标，通过在草稿纸上简单地勾勒就能够很方便地得到它的 4 种状态，4 种状态的 Coordinate 数组分别为：

```
//注意: Coordinate(列, 行)
private final Coordinate[] stateOne = {new Coordinate(0, 0), new Coordinate(0, 1),
```

```
        new Coordinate(0, 2), new Coordinate(1, 2)};
private final Coordinate[] stateTwo = {new Coordinate(0, 2), new Coordinate(1, 2),
        new Coordinate(2, 2), new Coordinate(2, 1)};
private final Coordinate[] stateThree = {new Coordinate(2, 2), new Coordinate(2, 1),
        new Coordinate(2, 0), new Coordinate(1, 0)};
private final Coordinate[] stateFour = {new Coordinate(2, 0), new Coordinate(1, 0),
        new Coordinate(0, 0), new Coordinate(0, 1)};
```

stateOne～stateFour所代表的状态依次对应了图11-19从左至右的4种形状，读者可以运行程序来确认这些坐标的正确性。在确定了这4种状态之后，transform()方法就很简单了，代码如下：

```
@Override
public void transform() {
    if(currentState == stateOne) currentState = stateTwo;
    else if(currentState == stateTwo) currentState = stateThree;
    else if(currentState == stateThree) currentState = stateFour;
    else if(currentState == stateFour) currentState = stateOne;
}
```

参照L类的写法，可以很快实现其余的分别代表I、J、S、T、Z形方块的类，为了更快地开发出游戏的原型，这里就先不实现这些形状的方块了，接下来先实现其他方面的功能。

5. 将方块放入游戏

前面已经封装好了O类和L类，它们分别代表了O形和L形方块，并且还提供了一些需要使用到的方法，这里将要实现的是将方块放入到游戏区域中，使得方块可以在游戏区域进行左、右、下的移动，并且能够进行变换。

首先，将方块放入到游戏区域中，考虑到俄罗斯方块游戏在进行时方块的到来是随机的，因此此处决定实现一个名为emergeOneTetrisObject的方法，该方法用于随机地产生一个方块，实际将新方块绘制出来的工作则交给另一个方法buildNewTetrisObject来完成。为此，需要为TetrisView增添一个成员变量，表示新到来的方块：

```
private static TetrisObject newComingTetris;
```

emergeOneTetrisObject()方法和buildNewTetrisObject()方法的代码如下：

```
//随机产生一个方块
private TetrisObject emergeOneTetrisObject(){
    TetrisObject newTetrisObject = new L();
    newTetrisObject.randomState();
    return newTetrisObject;
}
//将新方块放入游戏
private void buildNewTetrisObject(){
    newComingTetris = emergeOneTetrisObject();
    Coordinate[] newComingTetrisForm = newComingTetris.getCurrentTetrisState();
    for(int i = 0; i < newComingTetrisForm.length; i++){
```

```
12            setTile(newComingTetris.getTetrisColor(),
13                newComingTetrisForm[i].row + newComingTetrisRow,
14                newComingTetrisForm[i].column + newComingTetrisColumn);
15        }
16    }
```

可以看到,目前的 emergeOneTetrisObject()方法并不是随机产生的,因为目前还没有实现所有的方块类型,因此这里直接产生一个 L 形的方块。第 04 行是为 TetrisObject 接口新增的一个方法,因为考虑到新方块产生时它的状态也应该是随机的,L 类的 randomState()方法代码如下:

```
@Override
public void randomState() {
    Random random = new Random();
    int i = random.nextInt(4);
    switch (i) {
    case 0:
        currentState = stateOne;
        break;
    case 1:
        currentState = stateTwo;
        break;
    case 2:
        currentState = stateThree;
        break;
    case 3:
        currentState = stateFour;
        break;
    default:
        break;
    }
}
```

buildNewTetrisObject()方法的内容就是根据当前新产生的方块的状态,以(newComingTetrisRow,newComingTetrisColumn)为基准坐标,将方块绘制出来。

```
private final int newComingTetrisColumn = 4;
private final int newComingTetrisRow = 1;
```

最后需要做的就是在 surfaceCreated()方法中调用 buildNewTetrisObject()方法即可:

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
    ……
    clearTiles();
    buildNewTetrisObject();
    drawTheTiles();
}
```

再次运行示例,可以得到如图 11-29 所示的效果。

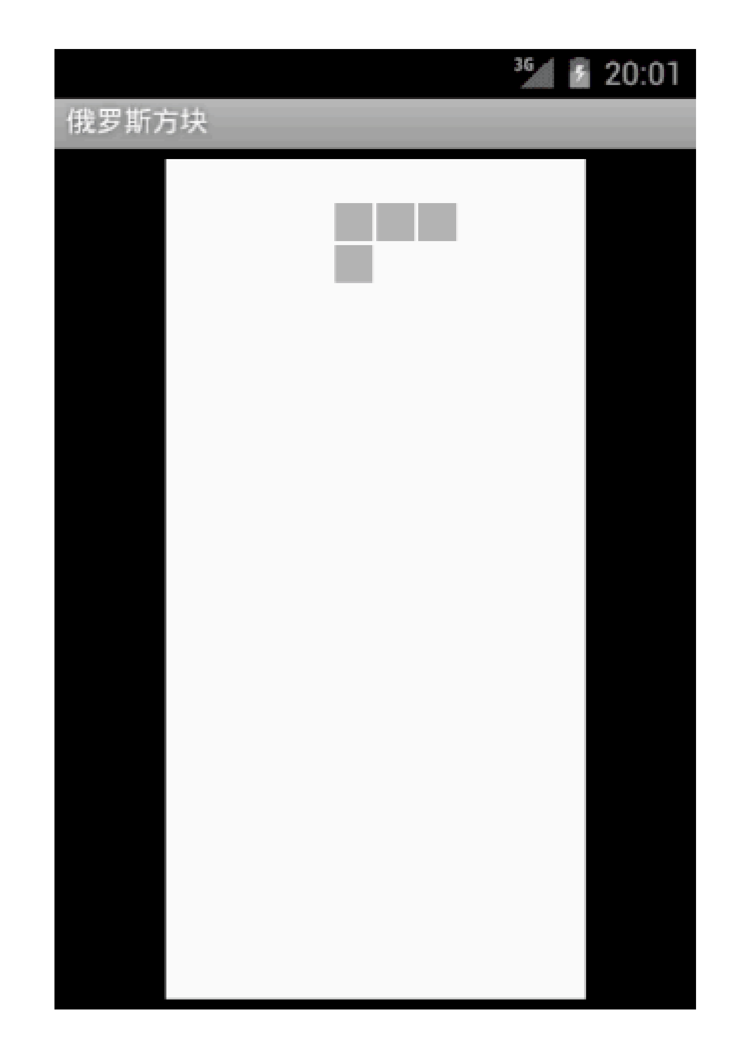


图 11-29　放入游戏区域的 L 形方块

接下来要实现的是对方块的操作功能,即实现对方块的移动和变换,这时就需要通过重写 onKeyDown()方法来实现对按键的响应,按照游戏需求,需要提供 3 个按键用于控制方块左移、右移和下移,还需要提供一个按键用于变换方块的状态,这里选择的是上下左右 4 个方向键。为此,onKeyDown()方法会是如下的框架形式:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
        //变换方块状态的代码
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
        //下移方块的代码
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
        //左移方块的代码
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
        //右移方块的代码
        return true;
    }
    return super.onKeyDown(keyCode, event);
}
```

考虑到在游戏中的任一时刻,能够操作的仅仅是当前处于活动状态的方块,因此有必要将这些处于活动状态的方块与不是处于活动状态的方块进行区分,为此,又加入了一个与 mTileArray 尺寸一样的二维数组 isTileActive,数组存放的是布尔型变量:

```
private int[][] mTileArray = new int[NUM_ROWS][NUM_COLUMNS];
 private boolean[][] isTileActive = new boolean[NUM_ROWS][NUM_COLUMNS];
```

相似地,为这个数组添加了如下几个配套方法:

```
//设置某个 Tile 为活动状态
private void setActiveTile(int row, int column) {
    isTileActive[row][column] = true;
}

//取消某个 Tile 的活动状态
private void unsetActiveTile(int row, int column) {
    isTileActive[row][column] = false;
}
//取消所有活动方块
private void unsetAllActiveTile(){
    for (int row = 0; row < NUM_ROWS; row++) {
        for (int column = 0; column < NUM_COLUMNS; column++) {
            unsetActiveTile(row, column);
        }
    }
}
```

在实现操作方块的方法时，只要维护好这两个二维数组的值就可以了。用于操作方块的方法命名如下：

- transformActiveTileArray()——变换方块状态的方法。
- leftShiftActiveTileArray()——左移方块。
- rightShiftActiveTileArray()——右移方块。
- downShiftActiveTileArray()——下移方块。

各方法中都包含的 ActiveTileArray 后缀正是为了强调操作的目标方块是处于 Active 状态的方块，下面依次实现这 4 个用于操作方块的方法。

首先来实现用于变换方块状态的 transformActiveTileArray()方法。要实现这个方法，不妨尝试一下使用伪代码编程的方式来组织代码，变换方块状态可以通过如下的步骤实现：

```
private void transformActiveTileArray(){
    获取当前处于活动的方块的状态;
    清除当前方块状态所影响到的游戏区域;
    变换方块到下一个状态;
    更改当前方块状态所影响到的游戏区域;
}
```

将上述伪代码转换为 Java 代码：

```
private void transformActiveTileArray(){
    Coordinate[] currentCoordinates = newComingTetris.getCurrentTetrisState();
      for(int i = 0; i < currentCoordinates.length; i++){
        isTileActive[currentCoordinates[i].row + basePoint.row]
                [currentCoordinates[i].column + basePoint.column] = false;
        unsetTile(currentCoordinates[i].row + basePoint.row,
                currentCoordinates[i].column + basePoint.column);
    }
    newComingTetris.transform();
    currentCoordinates = newComingTetris.getCurrentTetrisState();
      for(int i = 0; i < currentCoordinates.length; i++){
```

```
12             isTileActive[currentCoordinates[i].row + basePoint.row]
13                     [currentCoordinates[i].column + basePoint.column] = true;
14               setTile ( newComingTetris. getTetrisColor ( ), currentCoordinates [ i ]. row +
basePoint.row,
15                     currentCoordinates[i].column + basePoint.column);
16         }
17   }
```

其中，newComingTetris 是代表当前活动方块的对象：

```
private static TetrisObject newComingTetris;
```

上述代码的第 02 行、第 03～08 行、第 09～10 行和第 11～16 行依次对应了前面伪代码的 4 个步骤。可以看到，方法的关键就是对两个数组的操作，前面介绍的基准点(basePoint)在这里得到了使用，basePoint 被声明为 TetrisView 的成员变量，它的初始坐标即(newComingTetrisRow，newComingTetrisColumn)：

```
private static TetrisObject newComingTetris;
private final int newComingTetrisColumn = 4;
private final int newComingTetrisRow = 1;
private Coordinate basePoint = new Coordinate(newComingTetrisColumn, newComingTetrisRow);
```

其余 3 个方法的实现相对于 transformActiveTileArray()方法来说所需的操作要少一点，这是因为这 3 个方法对方块所产生的影响都是平移，因此两个数组的变化就比较有规律，例如左移就是让活动方块所包含的单元格坐标的列坐标整体减 1，右移则是让列坐标整体加 1，而下移则是让行坐标整体加 1。下面直接给出这 3 个方法的代码，需要注意的一点就是在各方法的结尾对 basePoint 的修改。

```
private void leftShiftActiveTileArray(){
    for (int column = 0; column < NUM_COLUMNS; column++) {
        for (int row = 0; row < NUM_ROWS; row++) {
            if(isTileActive[row][column]){
                unsetActiveTile(row, column);
                unsetTile(row, column);
                setActiveTile(row, column - 1);
                setTile(newComingTetris.getTetrisColor(), row, column - 1);
            }
        }
    }
    basePoint.column--;//列坐标减 1
}
private void rightShiftActiveTileArray(){
    for (int column = NUM_COLUMNS - 1; column >= 0; column--) {
        for (int row = 0; row < NUM_ROWS; row++) {
            if(isTileActive[row][column]){
                unsetActiveTile(row, column);
                unsetTile(row, column);
                setActiveTile(row, column + 1);
                setTile(newComingTetris.getTetrisColor(), row, column + 1);
```

```
                }
            }
        }
        basePoint.column++;//列坐标加 1
    }
    private void downShiftActiveTileArray(){
        for (int row = NUM_ROWS - 1; row >= 0; row--) {
            for (int column = NUM_COLUMNS - 1; column >= 0; column--) {
                if(isTileActive[row][column]){
                    unsetActiveTile(row, column);
                    unsetTile(row, column);
                    setActiveTile(row + 1, column);
                    setTile(newComingTetris.getTetrisColor(), row + 1, column);
                }
            }
        }
        basePoint.row++;
    }
```

可以看到，这 3 个方法非常相似，只有在处理活动方块的坐标时有着微小的差异。

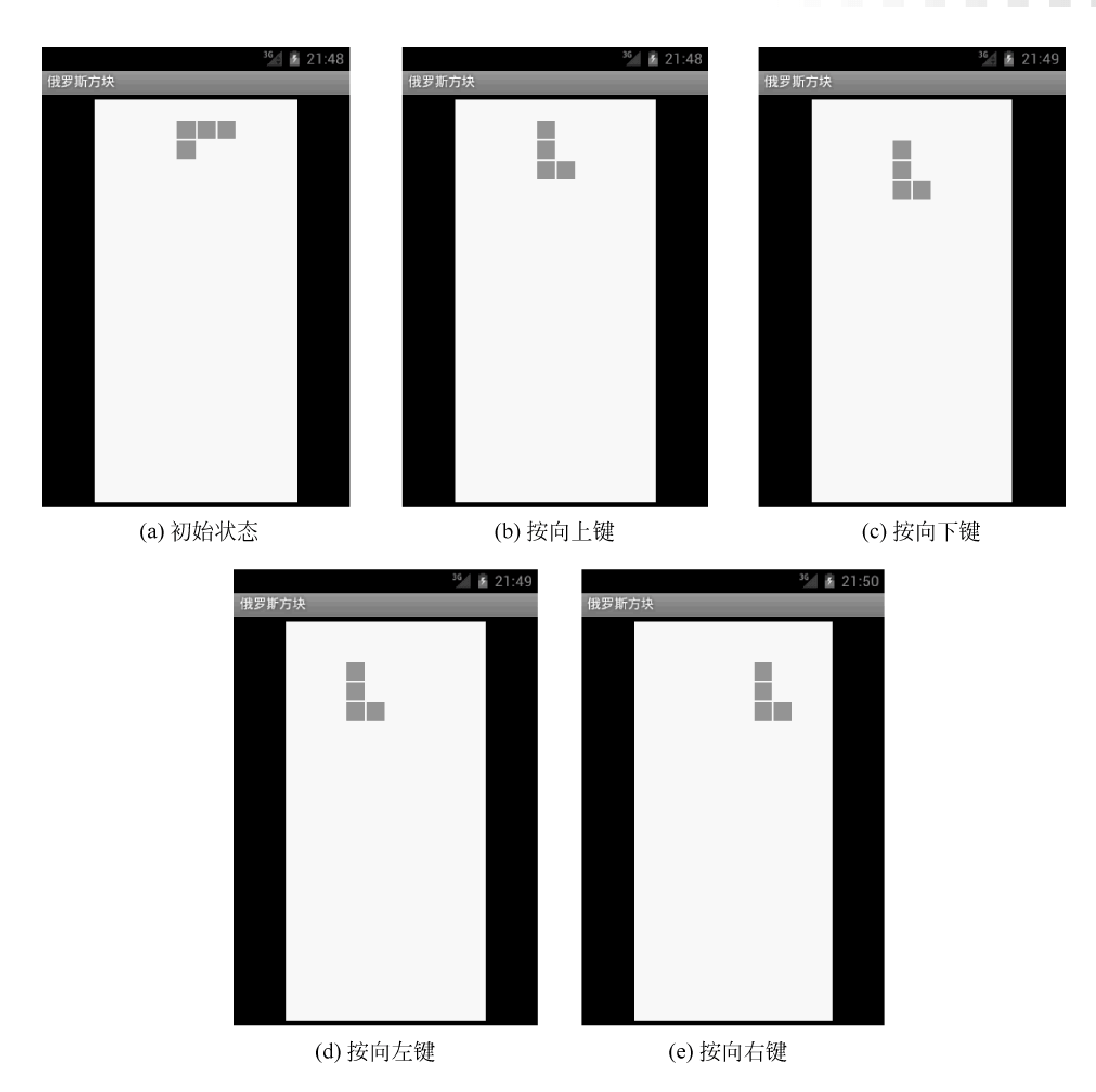
实现了用于操作的 4 个方法，将它们分别加入到 onKeyDown()方法中的对应位置，即可实现通过按键来触发相应操作方法的功能：

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
        transformActiveTileArray();
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
        downShiftActiveTileArray();
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
        leftShiftActiveTileArray();
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
        rightShiftActiveTileArray();
        return true;
    }
    return super.onKeyDown(keyCode, event);
}
```

至此，已经能够对被放入到游戏区域的 L 形方块执行 4 种操作了，效果如图 11-30 所示。

6. 游戏的驱动（一）

通常游戏都需要一个引擎来驱动，否则游戏就不能够向前运行，在前面分析 Snake 的时候就已经对这个概念进行了说明。俄罗斯方块游戏的引擎功能很简单，实际上就是使方块能够按照一定的频率下落即可，下面就借鉴 Snake 引擎的实现方式，来实现俄罗斯方块的引擎，判断引擎是否成功实现的标准就是能够使得在本节第 5 部分中实现的 L 形方块能够按一定的频率下落。



(a) 初始状态 (b) 按向上键 (c) 按向下键

(d) 按向左键 (e) 按向右键

图 11-30 各种按键的效果

下面再次给出在 Snake 中用于实现引擎的代码：

```
private RefreshHandler mRedrawHandler = new RefreshHandler();

class RefreshHandler extends Handler {

    @Override
    public void handleMessage(Message msg) {
        SnakeView.this.update();//每次收到消息,即执行更新方法
        SnakeView.this.invalidate();
    }

    public void sleep(long delayMillis) {
        this.removeMessages(0);
        sendMessageDelayed(obtainMessage(0), delayMillis);//睡眠 delayMillis 再次发消息
    }
};
```

此处需要修改的就是第 07 行调用到的 update()方法，根据俄罗斯方块游戏的特点将这个方法命名为 tick()，意为每经过一段时间即调用一次该方法，使方块能够自动下落。参照 Snake 的 update()方法，很容易写出 tick()方法：

```
private void tick(){
    downShiftActiveTileArray();
    drawTheTiles();
    mRedrawHandler.sleep(INTERVAL_BETWEEN_TICKS);
}
```

如上面的代码所示，只需要执行一次 downShiftActiveTileArray()方法，然后执行 drawTheTiles()方法更新显示即可让方块下落一行，然后再使用 mRedrawHandler 的 sleep()方法使得 tick()方法能够在一段时间后再次被调用，从而实现游戏的驱动。运行游戏，可以发现方块能够自动下落了，在不进行手动操作的情况下游戏的变化如图 12-31 所示。

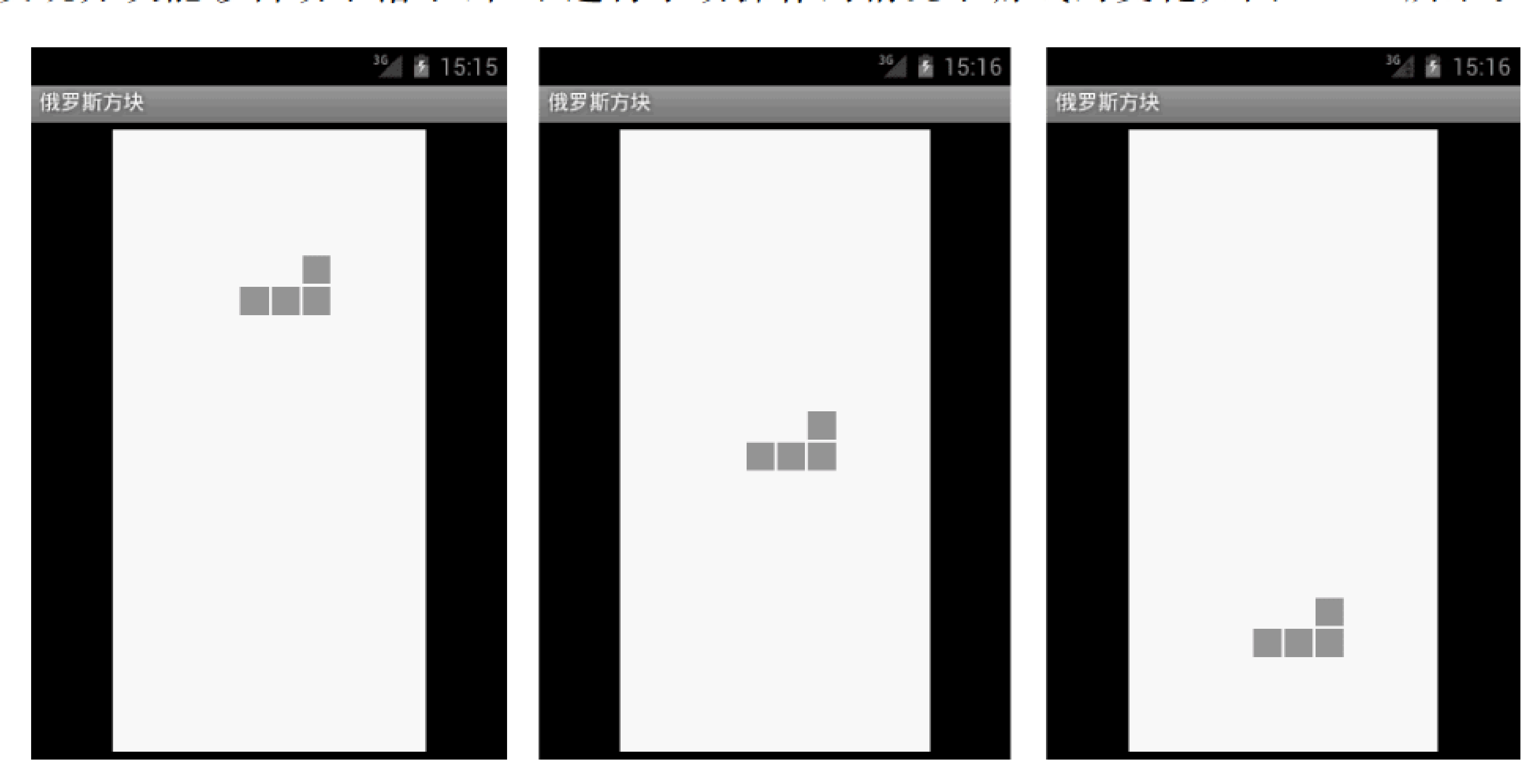


图 11-31　方块自动下落

然而，目前实现的这个驱动还不完整，完整的驱动需要能够使当前的活动方块在“落地”后变为非活动方块，同时在游戏区域上方放入新的活动方块并继续使其下落。在实现完整的驱动功能之前，需要解决一个问题，即如何判断方块“落地”这个问题。这就涉及游戏中的一个经典问题——碰撞检测。下面将着手解决碰撞检测问题，然后再继续完善游戏的驱动。

7. 碰撞检测

前面实现了方块的自动下落。细心的读者会发现，如果让游戏继续执行，当方块下落到最下面一行后，游戏会报错并且退出，到 LogCat 中进行查看会发现如下信息，截图如图 11-32 所示。

从 LogCat 输出的信息可以看出，这个错误是由数组下标越界所造成的，数组下标为何会越界呢？这是由于没有对游戏的一些边界情况进行处理。事实上，俄罗斯方块的游戏区域的四面都是不能够“穿越”的，相当于四面都是“墙壁”。事实上，不仅仅是当方块落到最下面一行会导致数组下标越界，如果操作方块左移或右移至边界同样会造成游戏的崩溃。要

threadid=1: thread exiting with uncaught exception (group=0x409961f8)
FATAL EXCEPTION: main
java.lang.ArrayIndexOutOfBoundsException: length=20; index=20
at com.segmac.tetris.TetrisView.setActiveTile(TetrisView.java:328)
at com.segmac.tetris.TetrisView.downShiftActiveTileArray(TetrisView.java:199)
at com.segmac.tetris.TetrisView.tick(TetrisView.java:133)

图 11-32 LogCat 截图

实现对这个情况的正确处理，就要依靠碰撞检测来实现。

游戏中的碰撞检测一直都是一个十分关键的部分，碰撞检测的合理性和精确度决定了游戏的真实度和可玩性，好的碰撞检测能够使得玩家操作的角色流畅地在地图上活动，而不会出现被卡在某个角落或者“掉落出”游戏的情况。一般地，2D 碰撞检测可以通过如下几种方式来实现(3D 也大致无异)：

- 将地图划分为单元格，然后根据两个物体相邻或者重叠来判断碰撞。
- 将物体都封装到一个最小的能够包含住这个物体的矩形内，然后根据矩形是否重叠来判断碰撞，为了使判断更精确，还可以将一个物体分为多个部分，每个部分由一个矩形封装，然后整体进行碰撞检测。
- 将物体封装到圆形内，原理与矩形一样，类似地，也可以使用其他形状来实现。
- 基于像素的检测，实际上可以理解为最精细的一种矩形碰撞检测。

上面介绍的几种碰撞检测原理，都是将游戏中的物体视为刚体来进行判断的，如果要进行更加真实的碰撞检测，通常需要借助于物理引擎来实现。物理引擎的作用就是将现实世界中的物理原理进行高度模拟，从而构造出一个高度仿真的虚拟环境，通过给物体赋予特定的物理特性，然后将碰撞检测交给物理引擎去处理即可。有关物理引擎的知识已经超出了本书的范围，有兴趣的读者可以到互联网查阅相关知识。

俄罗斯方块游戏虽然很简单，但它的实现也离不开对碰撞的检测。很明显，可以直接使用前面提到的第一种原理来进行碰撞检测，根据游戏分析，碰撞检测主要是为了实现对如下几个事件的判断：

- 方块是否已经贴上地面，即不能再向下移动。地面可以理解为已累积的方块的最上方轮廓，或者没有方块状态时的游戏区域下边缘。
- 方块是否能够向左或者右方移动。即左(右)方是否已经是区域边缘？或者左(右)方是否存在非活动状态的单元格？
- 方块是否能够进行变换。根据方块各个状态的坐标关系计算，如果方块变换到下一个状态，是否会与非活动状态的单元格发生碰撞？是否会造成方块部分越出游戏区域？

得到如上的一些需要判断的事件后，就可以开始编写代码了，将用于碰撞检测的方法命名为 testCollide()，由于碰撞检测需要在每个操作执行之前进行，即 testCollide()方法要在 4 个 ** ActiveTileArray()方法中调用，需要向 testCollide()方法传递当前操作的类型(左移、右移、下移或者变化)，对于不同的操作类型使用对应的碰撞检测方法。另外，在俄罗斯方块游戏的情景中，每一个方块都由 4 个单元格组成，因此需要独立的对每个单元格进行碰撞检测，所以还需要向 testCollide()方法传递当前检测的单元格坐标，testCollide()方法的

代码如下：

```
private boolean testCollide(int actionType, int column, int row){
    switch(actionType){
    case ACTION_TRANSFORM://操作类型为变换
        //得到当前活动单元格列表
        Coordinate[] nextCoordinates = newComingTetris.getNextTetrisState();
        for(int i = 0; i < nextCoordinates.length; i++){
            //变换会导致单元格越出上边界
            if(nextCoordinates[i].row + basePoint.row < 0) return true;
            //变换会导致单元格越出下边界
            if(nextCoordinates[i].row + basePoint.row >= NUM_ROWS) return true;
            //变换会导致单元格越出左边界
            if(nextCoordinates[i].column + basePoint.column < 0) return true;
            //变换会导致单元格越出右边界
            if(nextCoordinates[i].column + basePoint.column >= NUM_COLUMNS) return true;
            //变换会导致单元格与已存在的单元格发生碰撞
            if(!isTileActive[nextCoordinates[i].row][nextCoordinates[i].column]
                    && mTileArray[nextCoordinates[i].row][nextCoordinates[i].column]
                            != Color.WHITE) return true;
        }
        break;
    case ACTION_LEFTSHIFT://操作类型为左移
        //左移会导致单元格越出左边界
        if(column - 1 < 0) return true;
        //左移会导致单元格与已存在的单元格发生碰撞
        if(!isTileActive[row][column - 1] && mTileArray[row][column - 1]
                != Color.WHITE) return true;
        break;
    case ACTION_RIGHTSHIFT://操作类型为右移
        //右移会导致单元格越出右边界
        if(column + 1 > NUM_COLUMNS - 1) return true;
        //右移会导致单元格与已存在的单元格发生碰撞
        if(!isTileActive[row][column + 1] && mTileArray[row][column + 1]
                != Color.WHITE) return true;
        break;
    case ACTION_DOWNSHIFT://操作类型为下移
        //下移会导致单元格越出下边界
        if(row + 1 > NUM_ROWS - 1) return true;
        //下移会导致单元格与已存在的单元格发生碰撞
        if(!isTileActive[row + 1][column] && mTileArray[row + 1][column]
                != Color.WHITE) return true;
        break;
    default:
        break;
    }
    return false;
}
```

如上面的代码所示，方法根据操作类型来执行相应的检测代码，然后返回检测结果(true 为发生碰撞，false 为不发生碰撞)。注释中已经详细地对检测方法进行了说明，在实现了 testCollide()之后，需要在前面的 4 个 ** ActiveTileArray()方法中添加对其进行调用

的代码，例如需要在 transformActiveTileArray()方法中添加如下的第 02～05 行代码：

```
01  private void transformActiveTileArray(){
02      if(testCollide(ACTION_TRANSFORM, 0, 0)){ //进行变换类型的碰撞检测
03          Log.v(TAG, "Transform failed!");
04          return;
05      }
06      Coordinate[] currentCoordinates = newComingTetris.getCurrentTetrisState();
07      ……
08  }
```

这样，就能够在每次要执行变换方块操作之前，对这个操作的结果进行一个预判断，如果会发生碰撞，那么方法就直接返回而不执行变换方块的操作。类似地，可以在 leftShiftActiveTileArray()方法中添加如下的代码：

```
private void leftShiftActiveTileArray(){
    for (int column = 0; column < NUM_COLUMNS; column++) {
        for (int row = 0; row < NUM_ROWS; row++) {
            if(isTileActive[row][column]){
                if(testCollide(ACTION_LEFTSHIFT, column, row)) return;
            }
        }
    }
    for (int column = 0; column < NUM_COLUMNS; column++) {
    ……
}
```

rightShiftActiveTileArray()方法和 downShiftActiveTileArray()方法中需要添加的代码与之类似，只需要改变传入 testCollide()的 ACTION 类型。

再次运行示例，可以发现方块在接触边界时不会导致程序崩溃了，可以对 L 形方块进行随意的操作。

8. 游戏的驱动(二)

在实现了碰撞检测后，可以继续来完成游戏驱动的其余部分了。目前的情况是游戏区域内只存在一个 L 形方块，并且这个 L 形方块总是处于活动状态(即使已经落至地面)。为了实现正常的游戏逻辑，还需要为游戏驱动添加如下的功能：

- 判断当前的活动方块已经落地。
- 将该方块由活动状态转为非活动状态。
- 随机产生下一个活动方块。

如何判断方块已经落地呢？我们已经知道，由于游戏驱动的效果，方块在每一次 tick()到来时，会自动向下掉落一行，因此可以在 tick()方法中通过 downShiftActiveTileArray()方法是否成功执行(是否发生碰撞)来判断方块是否落地，为此，需要为 downShiftActiveTileArray()方法增加返回值：

```
private boolean downShiftActiveTileArray(){
    for (int row = NUM_ROWS - 1; row >= 0; row--) {
```

```
            for (int column = NUM_COLUMNS - 1; column >= 0; column -- ) {
                if(isTileActive[row][column]){
                    //检测到即将发生碰撞,下落操作不执行,返回 false
                    if(testCollide(ACTION_DOWNSHIFT, column, row)) return false;
                }
            }
        }
        ……
        //下落操作不会发生碰撞,正常下落,返回 true
        return true;
    }
```

然后再修改 tick()方法，利用 downShiftActiveTileArray()方法的返回值来判断方块落地事件并且执行相应的代码：

```
01  private void tick(){
02      boolean tetrisActive = downShiftActiveTileArray();
03      if(!tetrisActive){
04          //加入相应的处理代码,即: 改变当前方块状态并产生下一个方块
05      }
06      drawTheTiles();
07      mRedrawHandler.sleep(INTERVAL_BETWEEN_TICKS);
08  }
```

代码第 02 行定义的布尔型变量 tetrisActive 的意义就是当前方块是否落地（当 tetrisActive = true 时，表示方块未落地；当 tetrisActive = false 时，表示方块已落地），根据它的值来判断是否执行“改变当前方块状态并产生下一个方块”的操作，这个操作又包含如下几个步骤：

- 将当前的方块状态变为非活动。
- 判断是否跨行。
- 产生下一个方块。

在这 3 个步骤中，第一个步骤可以通过调用 unsetAllActiveTile()方法来实现，第三个步骤直接调用 buildNewTetrisObject()方法即可，第二个步骤需要单独实现。为此，实现了一个名为 collapse()的方法，方法代码如下：

```
private void collapse(int baseRow, int activeRows){
    Log.v(TAG, "baseRow = " + baseRow + "; activeRows = " + activeRows);
    for (int row = baseRow; row < baseRow + activeRows; row++) {
        boolean collapseThisRow = true;
        for (int column = 0; column < NUM_COLUMNS; column++) {
            if(mTileArray[row][column] == Color.WHITE) collapseThisRow = false;
        }
        if(collapseThisRow){
            for(int shiftRows = row; shiftRows > 0; shiftRows -- ){
                for (int column = 0; column < NUM_COLUMNS; column++) {
                    mTileArray[shiftRows][column] = mTileArray[shiftRows - 1][column];
                }
            }
```

```
            }
        }
    }
```

由于判断一行是否会垮掉需要通过扫描该行是否完全被方块铺满来实现，为了减少扫描的行数（通过扫描整个矩阵当然能够实现判断跨行的功能，但是那样会引入大量务必要的扫描），为 collapse()方法设计了两个参数 baseRow 和 activeRows，其中 baseRow 即当前 basePoint 的行号，而 activeRows 则由具体的 TetrisObject 提供，这里的原理是：只有新落地的方块所参与到的行才有可能垮。通过这两个参数可以使得需要扫描的范围缩减至 activeRows 行，而 activeRows 最大值也仅仅为 4（当方块为 I 形并且以竖直状态落地）。为了实现这个功能，为 TetrisObject 接口添加一个 getRows()方法，该方法返回一个整数，代表当前方块所能够影响的最大行数：

```
    public interface TetrisObject {
        ……
        public abstract int getTetrisColor();
        public abstract int getRows();
    }
```

以 L 形方块为例，它的 getRows()方法为：

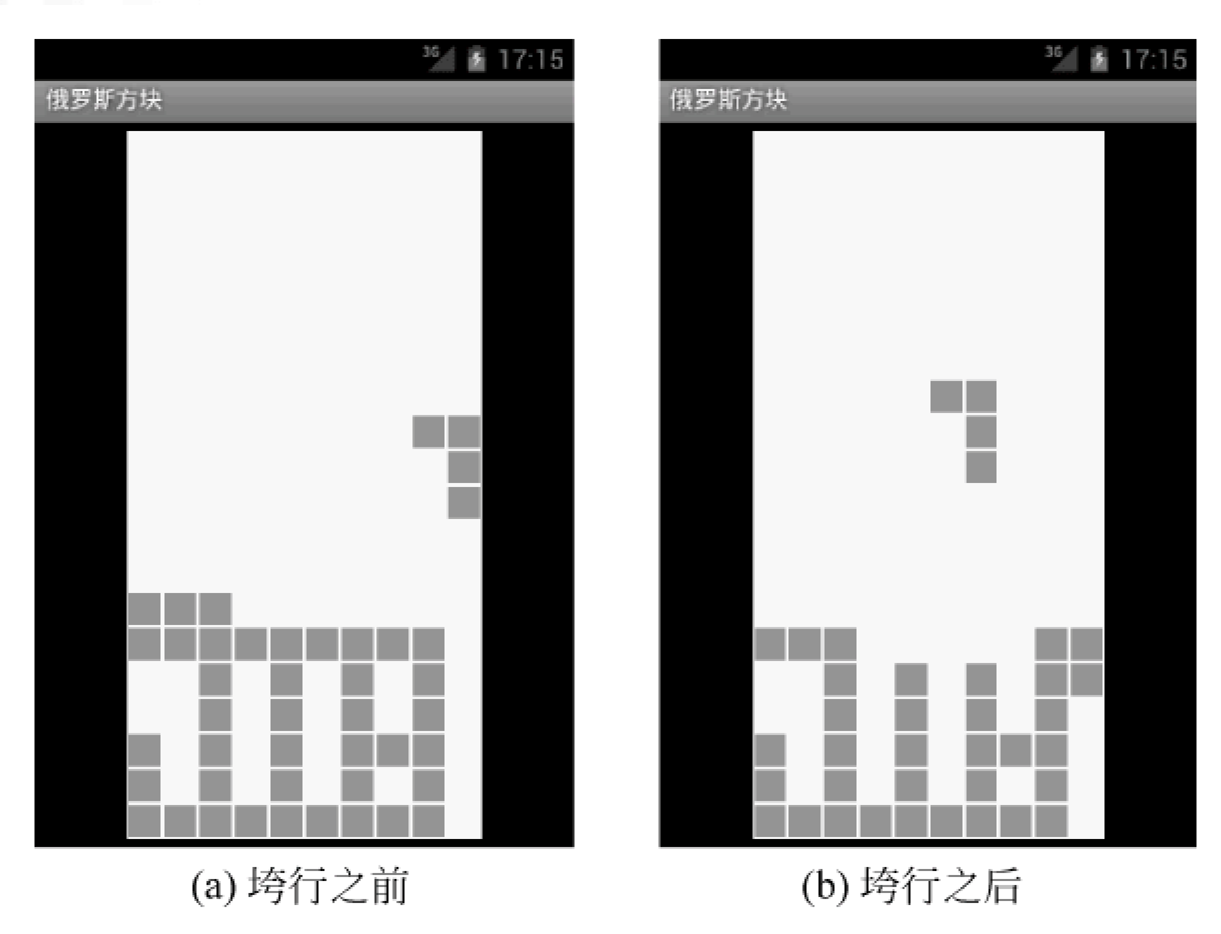
```
    @Override
    public int getRows() {
        if(currentState == stateOne) return 3;
        else if(currentState == stateTwo) return 3;
        else if(currentState == stateThree) return 3;
        else if(currentState == stateFour) return 2;
        return 0;
    }
```

即根据当前的方块状态来确定其影响的行数，这是一个简单的实现，实际上并不精确，例如当 L 形方块处于状态 2 时实际上只会影响相对于 baseRow 的第 2 行和第 3 行这两行，但是为了简便，仍然返回 3。

接下来可以完成 tick()方法：

```
private void tick(){
    boolean tetrisActive = downShiftActiveTileArray();
    if(!tetrisActive){
        unsetAllActiveTile();
        collapse(basePoint.row, newComingTetris.getRows());
        buildNewTetrisObject();
        basePoint = new Coordinate(newComingTetrisColumn, newComingTetrisRow);
    }
    drawTheTiles();
    mRedrawHandler.sleep(INTERVAL_BETWEEN_TICKS);
}
```

这样，一个更完整的驱动就已经完成了，再次运行程序，可以发现这已经是一个简易的俄罗斯方块小游戏了，虽然目前只存在一种 L 形状的方块。如图 11-33 所示。



(a) 垮行之前　　(b) 垮行之后

图 11-33　仅有 L 形方块的俄罗斯方块游戏原型

9. 游戏的完善

经过前面的一系列步骤，可以说已经完成了俄罗斯方块的核心部分，剩下的工作就是对其添砖加瓦进行完善了。鉴于这些功能的实现相对简单，为了节约篇幅，这里就仅对其进行简单的说明，读者可以通过阅读本书所附的完整代码来进行详尽的了解。

在前面实现的基础上，还需要对游戏进行如下一些完善：

- 实现其余 5 种形状的方块，并将 emergeOneTetrisObject()方法改为随机。
- 为游戏增添对下一个方块的预览功能。
- 为游戏增加状态控制功能，即可以暂停游戏(Pause)、结束游戏(Game Over)。
- 为游戏增加计分功能，并且根据一次性垮的行数给予不同的分数。

在本书所附的源码(见清华大学出版社网站)中包括了对上述功能的实现，除此之外，还可以为游戏增加音效、排行榜等功能。下面简要地对这 4 种功能的实现进行介绍。

1) 完成所有类型的方块

其余 5 种方块的实现直接参考 L 类的实现，需要注意的是确定好每种类型方块的几种状态的相对坐标(即 statusOne、statusTwo 等)，另外可以为每种类型方块赋予不同的颜色(getColor()方法)，还需要修改的就是 getRows()方法，其他的就基本没有什么变化了，在添加完成了方块之后，运行游戏，就能够得到如图 11-34 所示的效果。

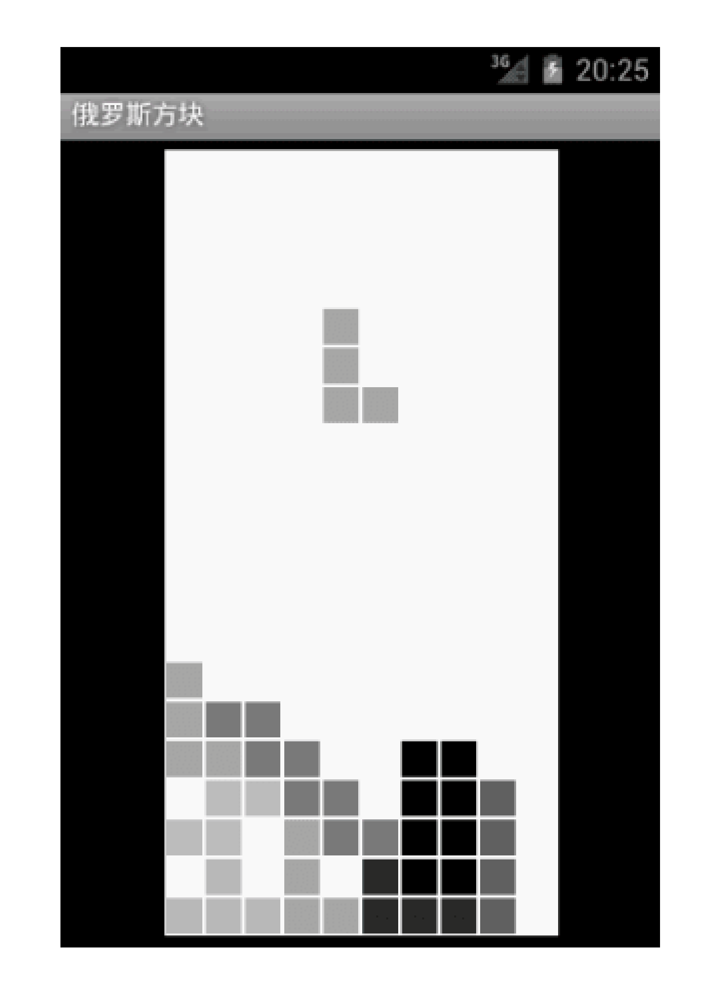


图 11-34　七种不同类型的方块

2) 方块预览功能

要为游戏增添对下一个方块的预览功能。首先需要在屏幕上开辟一个区域用于预览下一个到来的方块，由于游戏区域两侧还存在没有利用的空间，因此可以将这个预览区域添加到游戏区域的一侧，因为预览区域也会占用屏幕宽度，因此需要修改对 mTileSize、mXOffset、mYOffset 等参数的计算方法。引入如下一些变量，将预览区域设置为一个长和宽都为 5 个单元格的区域：

```
private static int SIZE_OF_PREVIEW = 5;//预览区域的尺寸
//方块预览区的左上角位置
private static int mPreviewXOffset;
private static int mPreviewYOffset;
//该数组代表了方块预览区域的方块
private int[][] mPreviewTileArray = new int[SIZE_OF_PREVIEW][SIZE_OF_PREVIEW];

private static TetrisObject previewTetris;
//预览方块的基准点坐标
private static final int PREVIEW_TETRIS_COLUMN = 1;
private static final int PREVIEW_TETRIS_ROW = 1;
```

事实上,可以将预览区域看成是另一个微型的游戏区域,从上面的变量定义也可以看出,它的实现也正是模仿了游戏区域的实现方式。新的游戏区域计算规则如下:

```
public void surfaceCreated(SurfaceHolder holder) {
    //得到当前屏幕的可用尺寸
    mHeightOfTheView = this.getHeight();
    mWidthOfTheView = this.getWidth();

    //得到每个方块的尺寸
    int tileMaxHeight = (int) Math.floor(mHeightOfTheView / NUM_ROWS);
    int tileMaxWidth = (int) Math.floor(mWidthOfTheView / (NUM_COLUMNS + SIZE_OF_PREVIEW));
    mTileSize = tileMaxHeight > tileMaxWidth ? tileMaxWidth : tileMaxHeight;

    //靠左显示游戏主区域
    mXOffset = 0;
    //用于在竖直方向上居中显示游戏主视图
    mYOffset = ((mHeightOfTheView - (mTileSize * NUM_ROWS)) / 2);

    //计算方块预览区的左上角位置
    mPreviewXOffset = mWidthOfTheView - SIZE_OF_PREVIEW * mTileSize;
    mPreviewYOffset = mYOffset;
}
```

然后再在 drawTheTiles()方法中添加用于绘制预览区域的代码即可。该段代码与游戏主区域的绘制代码基本一致:

```
private void drawTheTiles(){
    ......
    //绘制方块预览区域
    mCanvas.drawRect(mPreviewXOffset, mPreviewYOffset,
            mPreviewXOffset + SIZE_OF_PREVIEW * mTileSize,
            mPreviewYOffset + SIZE_OF_PREVIEW * mTileSize, mPaint);
    for (int row = 0; row < SIZE_OF_PREVIEW; row++) {
        for (int column = 0; column < SIZE_OF_PREVIEW; column++) {
            if (mPreviewTileArray[row][column] != Color.WHITE) {
                mCanvas.save();
                mCanvas.clipRect(mPreviewXOffset + column * mTileSize
                        + INTERVAL_BETWEEN_TILES,
                    mPreviewYOffset + row * mTileSize + INTERVAL_BETWEEN_TILES,
                    mPreviewXOffset + (column + 1) * mTileSize
                        - INTERVAL_BETWEEN_TILES,
```

```
                    mPreviewYOffset + (row + 1) * mTileSize - INTERVAL_BETWEEN_TILES);
                mCanvas.drawColor(mPreviewTileArray[row][column]);
                mCanvas.restore();
            }
        }
    }
    mSurfaceHolder.unlockCanvasAndPost(mCanvas);
}
```

实现了用于预览的区域后，需要改变一下新方块的产生方式，原来产生的新方块是直接产生并放入游戏主区域中，现在需要将新方块先放入到预览区域，然后主区域再从预览区域取出方块，同时预览区域生成下一个到来的方块，从而实现了方块预览功能。运行示例，将会看到如图 11-35 所示的效果。

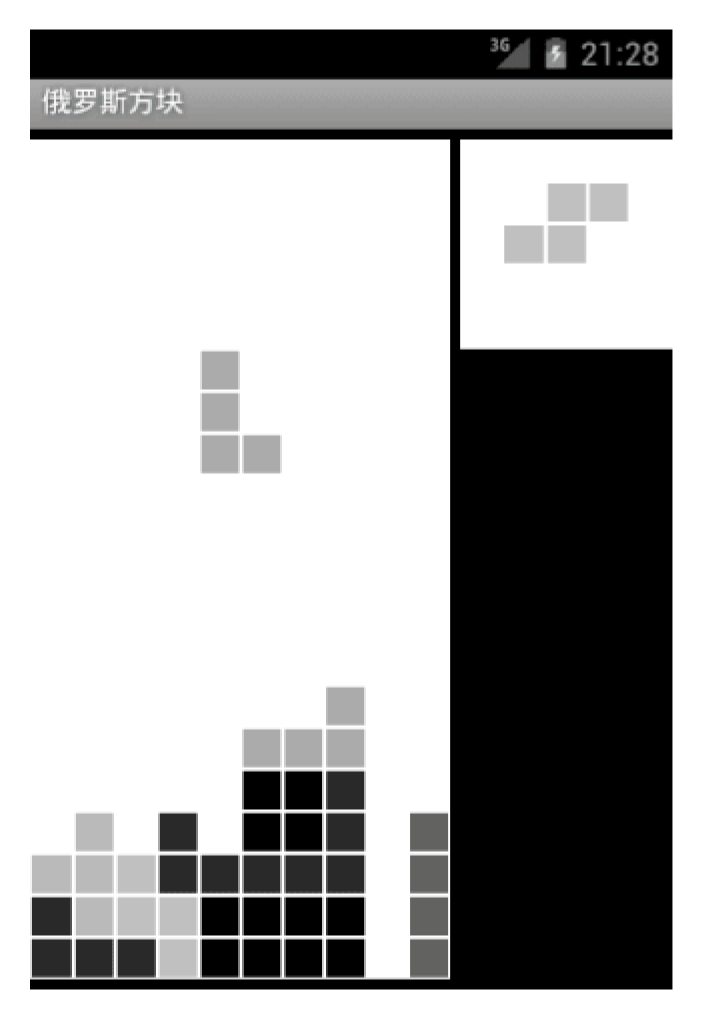


图 11-35 预览方块

3）游戏状态切换

接下来为游戏增加状态控制功能，这个可以完全参考 Snake 的实现方式，Snake 的实现使用了如下常量：

```
//游戏的状态,参考 Snake
private int mMode = READY;
public static final int PAUSE = 0;
public static final int READY = 1;
public static final int RUNNING = 2;
 public static final int LOSE = 3;
```

然后再实现一个 setMode()方法，在需要改变游戏状态时，只需要调用这个方法来改变游戏状态即可，具体的代码此处不再列出。这里需要注意的一点是，为了向用户提示当前的游戏状态（文字说明，例如 Snake 中的“Press Up To Play”），需要向视图中添加额外的控件（TextView），因此需要改变 Activity 的内容布局，在前面是直接通过如下的代码：

```
public class TetrisActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new TetrisView(this));
    }
}
```

直接将 TetrisView 设置为了 ContentView，这里需要使用 xml 文件来定义布局，并且将 TetrisView 作为一个标签添加到 xml 布局文件中，同样，可以参照 Snake 的实现，定义 xml 布局文件 main.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <com.segmac.tetris.TetrisView
        android:id="@+id/tetris"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

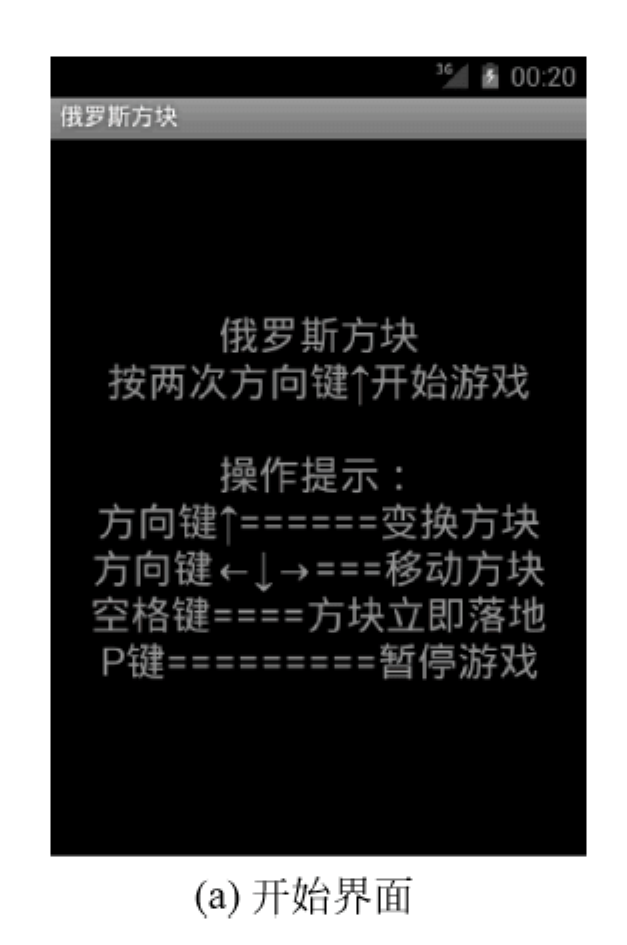
        <TextView
         android:id="@+id/game_status_text"
            android:text="@string/tetris_layout_text_text"
            android:visibility="visible"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerInParent="true"
            android:gravity="center_horizontal"
            android:textColor="#ff8888ff"
            android:background="@android:color/black"
            android:textSize="24sp"/>
    </RelativeLayout>
</FrameLayout>
```

然后将 Activity 修改为：

```
public class TetrisActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

即可。然后再在 TetrisView 中添加相应的 mStatusView()以及 setTextView()方法，在 TetrisActivity 中将 TextView 传递给 TetrisView 使用即可。

最终的实现效果如图 11-36 所示，这里还额外增加了一个使用 P 键暂停游戏的功能。



(a) 开始界面



(b) 按P键暂停游戏



(c) 游戏结束

图 11-36 游戏状态切换

其中游戏结束状态的判断是在 buildNewTetrisObject()方法中进行的：

```
private void buildNewTetrisObject(){
    newComingTetris = previewTetris;
    Coordinate[] newComingTetrisForm = newComingTetris.getCurrentTetrisState();
    for(int i = 0; i < newComingTetrisForm.length; i++){
        if(mTileArray[newComingTetrisForm[i].row + NEW_COMING_TETRIS_ROW]
                [newComingTetrisForm[i].column + NEW_COMING_TETRIS_COLUMN]
                    != Color.WHITE){
            setMode(LOSE);
            return;
        }
    }
    ……
}
```

实际上也是进行了一次较特殊的碰撞检测。

4）为游戏计分

俄罗斯方块游戏需要一个简单的计分系统，每当有新的层被垮掉时，玩家就会获得相应的分数，这里设计的得分规则如下：

- 垮 1 层——得 1 分。
- 垮 2 层——得 3 分。
- 垮 3 层——得 6 分。
- 垮 4 层——得 10 分。

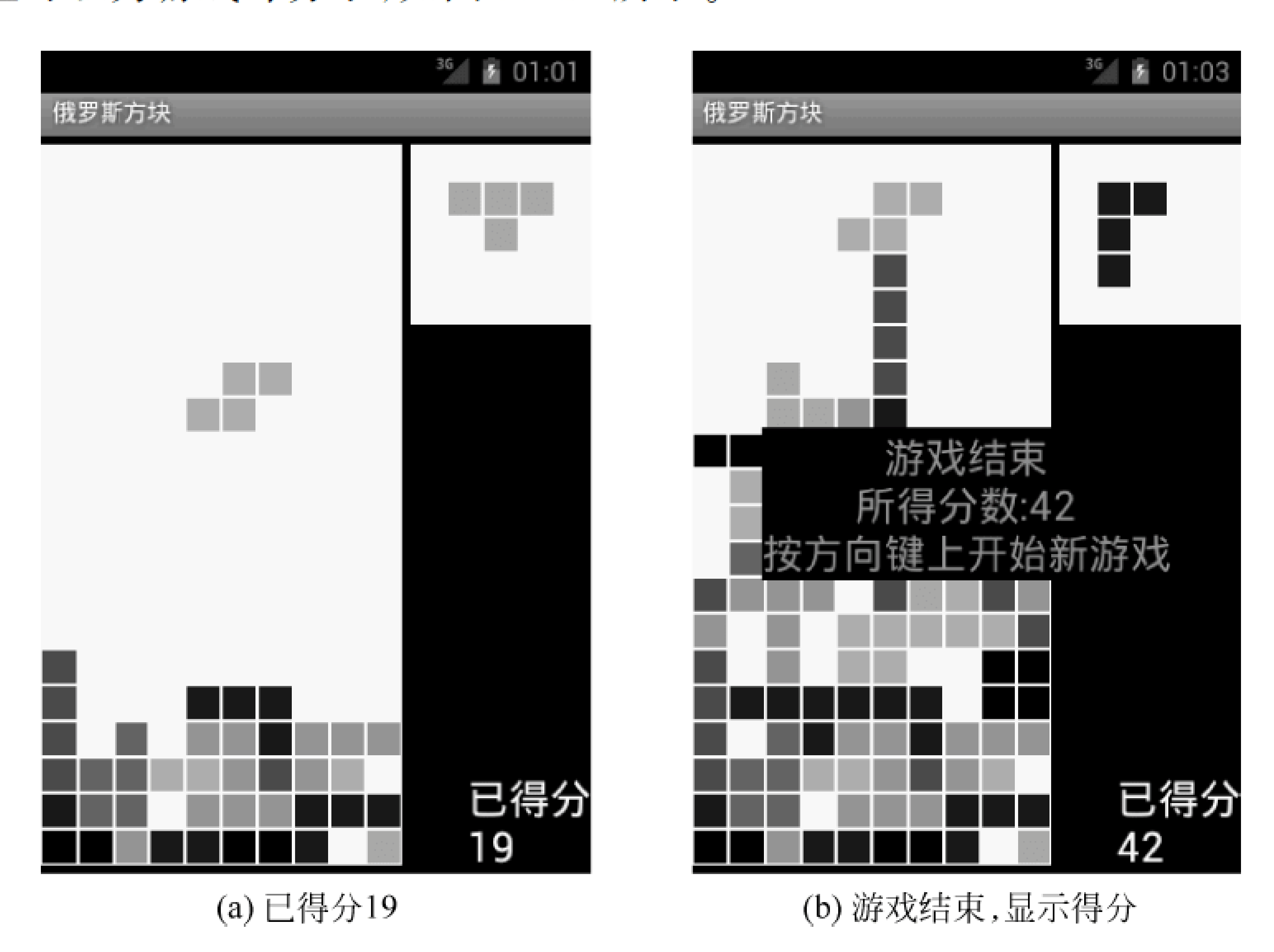
为了显示分数，还需要向布局中加入另外一个 TextView 控件，具体做法与前面添加用于描述状态的 mStatusView 一致。显而易见的，应该在 collapse()方法中进行相关的计分操作，有关代码如下：

```
private void collapse(int baseRow, int activeRows){
    int numCollapse = 0;
    for (int row = baseRow; row < (baseRow + activeRows < NUM_ROWS
          ? baseRow + activeRows : NUM_ROWS); row++) {
        boolean collapseThisRow = true;
        for (int column = 0; column < NUM_COLUMNS; column++) {
            if(mTileArray[row][column] == Color.WHITE) collapseThisRow = false;
        }
        if(collapseThisRow){
            for(int shiftRows = row; shiftRows > 0; shiftRows--){
                for (int column = 0; column < NUM_COLUMNS; column++) {
                    mTileArray[shiftRows][column] = mTileArray[shiftRows - 1][column];
                }
            }
            numCollapse++;
        }
    }

    //计分规则
    switch (numCollapse) {
    case 1:
```

```
22            mScore++;
23            break;
24        case 2:
25            mScore = mScore + 3;
26            break;
27        case 3:
28            mScore = mScore + 6;
29            break;
30        case 4:
31            mScore = mScore + 10;
32            break;
33        default:
34            break;
35        }
36        mScoreText.setText("已得分\n" + mScore);
37
38        //加速游戏规则,20 为加速间隔分数,1000 为初始间隔,100 为每升一级的增量
39        mLevel = ((mScore / 20) < 9) ? (mScore / 10) : 9;    //每 20 分升一级,最高 9 级
40        //每升一级,间隔减少 100,最快 100
41        intervalTime = 1000 - (100 * mLevel);
42    }
```

如上面的代码所示,numCollapse 用于统计一次垮下的行数,switch 选择器用于根据不同的行数计相应的分数,另外,第 38～41 行还设计了分数、等级与时间之间的关系。再次运行示例,已经可以为游戏计分了,如图 11-37 所示。



(a) 已得分19　　(b) 游戏结束,显示得分

图 11-37 已经添加了计分功能

5）继续完善游戏

至此,一个基本的俄罗斯方块游戏已经完成,不过仍然还有许多的功能可以添加,使得游戏更有趣,请读者发挥自己的想象力,继续完善这款经典的小游戏吧!

参考文献

1. 维基百科“游戏”词条：http://zh. wikipedia. org/wiki/游戏.
2. Snake 游戏深入解析：http://www. moandroid. com/?p=474.
3. 维基百科“俄罗斯方块”词条：http://zh. wikipedia. org/zh/俄罗斯方块.

第12章 Chrome 扩展

12.1 Chrome 简介

12.1.1 Chrome 的产生

Chrome 浏览器借鉴了一些源自苹果的 WebKit 和 Mozilla Firefox 的技术，并且有一个强大的 JavaScript 引擎——V8，这使得该浏览器在性能上比 IE 要略胜一筹。

WebKit 是苹果公司的浏览器 Safari 的核心，而 Safari 是 Apple 用户中最受欢迎的浏览器。Gecko 是 Mozilla Firefox 浏览器使用的内核代号，使用 Gecko 内核的浏览器也有不少，如 Netscape。这两个内核引擎在 12.2.1 节中有详细介绍。

WebKit 和 Gecko 都为开源项目，同样一个很流行的浏览器 Opera 使用内核 Presto，这是目前公认网页浏览速度最快的浏览器内核。

Chrome 的网上应用店如图 12-1 所示，可以根据需要找到有用的好玩的东西。网址为：https://chrome.google.com/webstore。



图 12-1　chrome 网上应用店

12.1.2 Chrome 的优势

Chrome 虽然属于浏览器中的后起之秀，但是凭借其一系列的优势，已经迅速地获取了大量的用户，到目前为止已经成为与 Firefox 的市场占有量不分伯仲的仅次于 IE 浏览器的第二大浏览器，简单来说，它具有如下的一些优势：

- 界面简洁，能够吸引大部分的年轻用户，Chrome 浏览器布局相对简洁、流畅，这对于上网本和其他小型便携式计算机用户来说，极具吸引力。一旦进入移动设备领域，Chrome 的高速浏览性能将确保其成为移动浏览器领域的佼佼者。
- 网页加载速度快，至少在视觉上给用户带来很快的网页加载速度。
- 稳定性高，Chrome 浏览器与其他浏览器相比，其稳定性更高，浏览器的每个标签都在相对独立环境中运行，每个标签就是一个进程，即使一个网页崩溃，其他的网页也不会受到影响，确保整个浏览器不至于瘫痪。Chrome 浏览器采用的这一设计模式，可以方便用户检查每个标签网页所占用的内存大小和处理器空间大小。
- 内置任务管理器，在浏览器的标签页栏中右击，打开任务管理器，或者用 shift＋esc 快捷键打开浏览器的任务管理器，可以查看浏览器中已打开的标签页和已安装的扩展的内存、CPU、网络、FPS 消耗情况。有经验的扩展开发人员还可以编写相应的内存消耗控制扩展来对其进行管理。
- 网页调试器，在 Chrome 浏览器中按快捷键 F12，即可以启动浏览器的网页调试器，其功能作用与 Firefox 中的 firebug 插件相类似，不同之处在于 firebug 是 Firefox 中的一个插件，需要安装才能使用，Chrome 浏览器中自带网页调试器，使得前端开发人员使用起来更方便。
- 越来越丰富的插件支持，Chrome 网上应用商店中的应用数量越来越大。
- 跨操作系统，能够在 Windows、Mac OS X、Linux 甚至 Android 上使用。
- 与 Google 所提供的其他应用高度整合，使人感觉用 Google 的产品就应该使用 Chrome 浏览器才能够得到最佳的体验，例如用户群庞大的 Gmail 应用，这能够增加一部分用户，使用 Google 账户对用户的一些数据（书签、日历、联系人）进行同步，只要有 Chrome 浏览器，就能够让用户在任意的计算机上使用自己的数据。
- 更新速度极快，借助 Google 强大的开发能力，Chrome 浏览器从发布以来就没有放慢过其更新的脚步，每一次更新都能够为用户带来更安全、更流畅的浏览体验，以及更加丰富的功能。

12.1.3 扩展的概念

1. Chrome Extension 是什么

一个 Chrome 扩展是由 HTML、CSS、JavaScript、图片等文件压缩而成。扩展实际上就是一个 Web 页面，可以用任何浏览器提供给 Web 页面的接口，从 XMLHttpRequest 到 JSON，再到 HTML 本地缓存都可以使用。

2. Chrome 扩展能做什么

我们会发现有些扩展在 Chrome 地址栏右侧区域增加一个图标，有些扩展能够和浏览器的一些元素(如书签、tab 导航标签)交互，有些扩展还可以和 Web 页面交互，甚至是从 Web 服务器获取数据。

每个扩展都是一个.crx 文件(类似 zip 的压缩文件)，由下列文件组成：

- 一个 manifest 文件(主文件，json 格式)。
- 至少一个 HTML 文件(主题可以没有 HTML 文件)。
- JavaScript 文件(可选，非必需)。
- 任何其他需要的文件(比如图片)。

3. 在开发扩展时可能使用到的技术

- HTML、CSS 和 JavaScript。
- chrome. * API。
- HTML5。
- <canvas>。
- <audio>。
- localStorage。
- Web Database。
- XMLHttpRequest。
- WebKit API 和 V8 API(JavaScript 引擎)。

12.2 Chrome 与 Firefox 的比较

12.2.1 Chrome 与 Firefox 的内核比较

引擎不仅是浏览器最重要的特性，同时也关乎浏览器对扩展的支持程度。我们已经知道，Chrome 所使用的是源自苹果的开源引擎 Webkit 以及 Javascript 引擎 V8，而这个 V8 确实不负众望，在各种测试中遥遥领先于其他的浏览器。Firefox 使用的是已经几乎变成 Mozilla 自行开发的 Gecko。

Webkit 是由苹果主推的一款开源的轻量级网页渲染引擎。Webkit 的主要特点为渲染速度极快、源码结构清晰、出色的标准(W3C)支持和低内存消耗等，如今已经被多款浏览器所使用。主要代表作品有 Safari 和 Chrome。它的足迹不仅遍及所有操作系统，而且还有手机，目前塞班 S60、WebOS、iPhone 和 Android 的默认浏览器均使用 Webkit 作为其内核。

Gecko 是一套开放源代码的、以 C++编写的网页排版引擎。目前为 Mozilla 家族网页浏览器以及 Netscape 6 以后版本浏览器所使用。这软件原本是由网景通讯公司开发的，现在则由 Mozilla 基金会维护。它的最大优势是跨平台，能在 Microsoft Windows、Linux 和 MacOS X 等主要操作系统上运行，它还提供了一个丰富的程序界面以供互联网相关的应用

程序使用,例如网页浏览器、HTML 编辑器、客户端/服务器等。

Google 会选择 Webkit 作为 Chrome 的核心渲染引擎,其核心的理念也就是:保持浏览器的简洁高效,并忠实地还原网页的最初效果。

相对而言,Gecko 就没有 Webkit 这么走运。Gecko 一直以来都以其臃肿庞大而遭人诟病,直到最近的版本,也就是 Firefox 3 开始才逐渐有了明显的改变。但 Gecko 因为其相对复杂的代码,始终没能彻底改变其内存使用和效率均高于 Webkit 的现状。

比起 Webkit,Gecko 一个最大的优势就在于它支持了更多的内容。Webkit 仅能用于进行 HTML 的渲染工作,而 Gecko 在 HTML 渲染之外,还提供了一种基于 XML 的用户界面生成引擎:XUL。XUL 被广泛应用于 Firefox 中,用来生成各种不同的 UI 界面,也包括各种主流的扩展。同样得益于 XUL 的应用,我们才可以轻而易举地将 Firefox 的外观改为任何样子。

12.2.2 页面加载过程对比

在启动上,Firefox 的初始启动速度要慢于 Chrome 浏览器,这个也并不是必然的,还是与浏览器中安装的扩展数有关。Firefox 在启动的过程中不会一直保持扩展的运行,只有当使用该扩展的时候才会运行,而 Chrome 在启动的初期就会加载扩展,并且在浏览器运行的整个过程中一直保持扩展的运行,这保证了用户使用扩展时的快速度,但是从内存占用的角度上来讲是消耗内存的。由于 Chrome 在启动的过程中也同时运行了扩展,所以安装的扩展越多,启动速度越慢。

12.2.3 扩展性对比

- 扩展安装时,Firefox 需要重启,而 Chrome 直接启用,无须重启。
- 扩展的数量两者已经十分接近,但从扩展的质量来看,Chrome 仍旧落后于 Firefox。
- Firefox 的扩展是融合到浏览器主进程中的,安全性不够,Chrome 扩展则是运行在隔离的进程中,单个扩展的安全问题不会影响到整个浏览器。
- FireFox 的扩展在启动时升级(需要用户确认),而 Chrome 扩展自动升级。

12.2.4 对浏览器的性能影响

- 在数量较少的扩展情况下,两者体验无区别,Chrome 可能稍占优势,因为独立进程在单个扩展崩溃时不会影响整个浏览器。
- 在数量中等,如 10~15 个扩展的情况下,Chrome 明显占有优势。
- 在数量较多的情况下,如 20 个扩展或更多,会发现 Firefox 的资源占用和速度跟中等数量的情况下没有太大区别,但是 Chrome 的内存占用节节攀升,打开浏览器的加载速度和扩展的增加数成正比,对于这一点,Chrome 如果能在以后的版本更新中进行改进就更好了。

12.2.5 扩展数比较

Chrome:截至 2011 年 9 月 16 日,Chrome 的应用扩展数已达到 12 873 个。

Firefox：截至 2011 年 9 月 16 日，Firefox 的应用扩展数超过 12 000 个。

可以看到，Chrome 虽然推出的时间很短，但是在扩展数上已经紧跟 Firefox，并有超越之态。

12.2.6 内存消耗

浏览器的内存占用过高和安全问题一直存在，在传统的浏览器中一个浏览器是一个进程，一个 tab 的崩溃会导致整个浏览器停止工作，在一个进程空间下用户的安全也得不到相对高的保护，Chrome 做了一个创新，每个 tab 是一个进程，采用 session 共享的机制来进行同一网站内的不同 tab 间的协调，而这样又带来的较大的内存占用。解决这两者间的矛盾是最后要完成的工作，但是凡事都要从头开始。

Firefox 与 Chrome 这两款浏览器中使用着不同的 Javascript 引擎和排版引擎，其中老版本的 Firefox 使用 Spidermonkey Javascript 引擎和 Layout(Gecko)引擎，而在 Chrome 中使用的是从开源项目 Chromium 中遗传来的 V8 Javascript 引擎并且利用的开源的 Webkit 来作为其排版引擎。

浏览器对与内存的使用主要有两个方面：

- 一是 Javascript 引擎为 Javascript 对象分配的内存空间。
- 二是在排版引擎中对 html 页面的存储。

在设计理念上，Chrome 和 IE 8 一样，都考虑到了将来的硬件。

Chrome 采用多标签浏览模式，可以隔离崩溃的标签并保护 Gmail 或谷歌文档套件等复杂的应用程序，这意味着总要比 Firefox、IE 7 等单进程模式浏览器使用更多的内存。这两种模式孰优孰劣，尤其是在目前的硬件环境下，还有待观察。

遵守以下 3 个规则，可解决 Chrome“吃内存”的问题：

- 使用 Userscript。
- 使用 Chrome 自带的 Flash 插件。
- 使用 Javascript 书签。

12.3 Chrome 扩展组件介绍

12.3.1 Chrome 扩展插件入门

先来看一个很简单的例子，实现扩展版的 Helloworld，需要的准备工具包括：

- 记事本，用来编写代码。
- Chrome 浏览器，这个必不可少。Windows 下，所有版本的 Chrome 都可以制作插件。Linux 下需要下载 Beta 版本，Mac 下载 dev 版本。

开始制作扩展版的 Helloworld。

(1) 在计算机中创建一个目录如 example13 来存放插件代码。

(2) 在目录里面创建文件 manifest.json(注意后缀名是.json，json 格式的详细介绍请见 12.3.2 节)，用记事本打开，写入如下代码：

```
{
 "name" : "Helloworld",
  "version": "1.0",
  "description": "我的第一个 Chrome 插件,helloworld",
  "browser_action":
 {
    "default_icon": "helloworld.gif"
  }
}
```

(3) 安装插件的方法。

① 点击右上角扳手,选择扩展程序,打开扩展中心。

② 点击右上角的“开发人员模式”,使得前面的“+”变成“-”,打开相应的菜单。如果一开始就是“-”,那么不用点击。

③ 点击“载入正在开发的扩展程序”按钮,导入刚才创建的文件夹。

如果一切顺利,Chrome 地址栏将会有个新图标,第一个插件便诞生了。

当然,在上面的例子中,smile.gif 与 icon.gif 是直接放在 example13 文件夹中的,但是随着以后开发的插件功能越来越多,开发人员也增多的时候,像上面那样把所有插件需要的文件都放在同一个文件夹内就很不方便。此时就可以在 example13 文件夹里再新建一个文件夹 img,专门用来存放插件需要的图片。相应地,manifest.json 中 defualt_icon 的值也要改成该图片的相对地址。

```
"default_icon": "/img/helloworld.gif"
```

至此,第一个插件完成了,可是虽然看起来挺像个插件,但是事实上它什么都做不了,所以,下面给它增加一些功能,让它也变得实用。

在 manifest.json 文件中的 browser_action 属性中添加 popup 动作,它将实现弹窗的作用,popup 的值 popup.html 即为弹出的框的样子。将 popup.html 存放在 example13 文件夹中,icon.gif 与 google.gif 放在 img 文件夹内。

```
"browser_action": {
    "default_icon": "icon.gif",
    "popup": "popup.html"
  }
```

popup.html 文件内容为:

```
<html>
<p>Hello,Chrome!</p>
<p><a href="http://www.google.com" target="_blank">你好,谷歌</a>
<p><img src="/img/google.gif" /></p>
</html>
```

完成之后回到 Chrome 的扩展管理页面,“重新载入”,现在点击插件图标看看,第一个插件制作已经成功了。

（4）打包插件。

如果想把自己制作的插件分享给其他人用，那么就需要将插件打包，打包步骤如下：

① 回到 Chrome 的插件扩展中心，单击“打包扩展程序”按钮。

② 选择刚才创建的文件夹，单击“确定”按钮生成后缀为 crx 和 cpm 的文件各一个。

做完第一个 helloworld 扩展之后，来看看整个程序中用到了哪些扩展中常用的页面呢？manifest 文件和 html 文件。

- manifest 文件。

主文件取名 manifest.json，用来描述这个扩展，包括扩展名字、版本、描述、权限等信息。下面是个典型的 manifest 文件，这个扩展可以调用 google.com 的内容。

```
{
  "name": "My Extension",
  "version": "2.1",
 "description": "Gets information from Google.",
  "icons": { "128": "icon_128.png" },
  "background_page": "bg.html",
 "permissions": ["http://*.google.com/", "https://*.google.com/"],
  "browser_action": {
 "default_title": "",
 "default_icon": "icon_19.png",
 "popup": "popup.html"
 }
}
```

绝大部分扩展有 background 文件，一个不可见的文件控制着整个扩展的运行。这个浏览器行为扩展的 background 文件是用一个 HTML 文件定义的（background.html），这个 background 文件中有 JavaScript 代码控制整个浏览器的活动。

- HTML 页面。

background 不是唯一存在的 HTML 文件，比如浏览器行为可能是弹出一个小窗口，这个小窗口的内容就可以调用一个 HTML 文件。Chrome 扩展也能够用 chrome.tabs.create()或 window.open()函数来显示 HTML 文件，这两种方式的显示不同之处在于：前者是在已打开的浏览器中创建新的选项卡，后者是打开新的窗口。扩展里面的 HTML 文件可以互相访问对方的 DOM 结构，可以引用其他文件中定义的函数。

下面对 chrome 扩展组件进行详细的介绍。

12.3.2 Manifest 文件介绍

1. 字段摘要

以下字段为 manifest.json 的字段，其中 name 和 version 是必需的，表示扩展名称，扩展版本号。

```
{
 //必需的字段
  "name": "My Extension",
```

```
    "version": "versionString",
    // 建议存在的字段
    "description": "A plain text description",
    "icons": { ... },
    "default_locale": "en",
    // 根据代码情况选定一个或没有
    "browser_action": {...},
    "page_action": {...},
    "theme": {...},
    // 添加你需要的
    "background_page": "aFile.html",
    "chrome_url_overrides": {...},
    "content_scripts": [...],
    "key": "publicKey",
    "minimum_chrome_version": "versionString",
    "options_page": "aFile.html",
    "permissions": [...],
    "plugins": [...],
    "update_url": "http://path/to/updateInfo.xml"
    "incognito": "split or spanning",
  }
```

2. 字段说明

1）description

描述，值为文本字符串（不是 HTML 或者其他格式，注意不能超过 132 个字符），用以描述该 extension 程序。

2）icons

Extension 程序的图标可以有一个或多个。至少提供两个大小的图标——48×48 和 128×128。48×48 的图标用在 extensions 的管理界面（在浏览器地址栏中输入 chrome://extensions 即可访问，同理，访问 chrome 的历史记录输入 chrome://history），128×128 的图标用在安装 extension 程序的时候。还可以指定一个 16×16 的图标当作 extension 的页面图标。

图标一般为 PNG 格式，因为有透明度的支持，不过浏览器引擎 WebKit 支持任何图片格式，包括 BMP、GIF、ICO 和 JPEG 等。下面是一个指定的图标的例子：

```
  "icons": {  "16": "icon16.png",
            "48": "icon48.png",
            "128": "icon128.png" }
```

注意：以上编写的图标不是固定的。随浏览器环境的改变而改变。例如，安装时弹出的对话框变小。

3）default_locale

默认的语言环境。

4）key

在开发程序加载完后，key 值可用在控制唯一的 ID。

5）minimum_chrome_version

需求的最小 Chrome 浏览器版本号。

6）name

值为纯文本字符串(不超过 45 个字符)，扩展的标识。该名称用在安装对话框中，Extension 的管理界面。

7）permissions

值为一个数组。每个权限可以是一个已知的字符串列表如标签，或一个匹配模式，可以访问一个或多个主机。

以下是 manifest 文件部分权限的例子，更多权限介绍请参看 13.3.6 节。

```
"permissions": [
  "tabs",
  "bookmarks",
  "http://www.blogger.com/",
  "http://*.google.com/",
  "unlimitedStorage"
]
```

8）version

1～4 个以点分隔的整数标识版本。一些应用于整数的规则：它们必须在 0～65 535，包括非零整数。例如，99 999 和 032 都是无效的。

下面是有效版本的一些例子：

(1)“版本”：1。

(2)“版本”：1.0。

(3)“版本”：2.10.2。

(4)“版本”：3.1.2.4567。

比较自动更新的系统版本，以确定是否已安装扩展需要更新。如果发布了的扩展是已安装扩展较新版本的字符串，扩展名自动更新。

比较开始从最左边的整数。如果这些整数是相等的，右边的整数进行比较，以此类推。例如，1.2.0 是一个比 1.1.9.9999 更新的版本。

一个缺少整数等于零。例如，1.1.9.9999 版本比是 1.1 更新的版本。

3. JSON 文件格式简介

JSON(Javascript Object Notation)是一种轻量级的数据交换语言，以文字为基础，易于让人阅读。尽管 JSON 是 Javascript 的一个子集，但 JSON 是独立于语言的文本格式，并且采用了类似于 C 语言家族的一些习惯。

JSON 格式是 1999 年 *JavaScript Programming Language*，*Standard ECMA-262 3rd Edition* 的子集合，所以可以在 JavaScript 以 eval()函数(JavaScript 通过 eval()调用解释器)读入。不过这并不表示 JSON 无法使用于其他语言，事实上几乎所有与网页开发相关的语言都有 JSON 库。

JSON 用于描述数据结构，有以下形式：

- 对象（object）——一个对象以“{”开始，并以“}”结束。一个对象包含一系列非排序的名称/值对，每个名称/值对之间使用“，”分区。
- 名称/值对（collection）——名称和值之间使用“：”隔开，一般的形式是：{name：value}一个名称是一个字符串；一个值可以是一个字符串、一个数值、一个对象、一个布尔值、一个有串行表或者一个 null 值。
- 值的有串行表（Array）——一个或者多个值用“，”分区后，使用“[”、“]”括起来就形成了这样的列表，形如：[collection，collection]字符串：以" "括起来的一串字符。
- 数值：一系列 0～9 的数字组合，可以为负数或者小数。还可以用“e”或者“E”表示为指数形式。
- 布尔值：表示为 true 或者 false。

在很多语言中，它被解释为阵列。

JSON 的格式描述可以参考 RFC 4627。

4. JSON 格式与 XML 格式的对比

JSON 与 XML 最大的不同在于 XML 是一个完整的标记语言，而 JSON 不是。这使得 XML 在程序判读上需要比较多的时间。主要的原因在于 XML 的设计理念与 JSON 不同。XML 利用标记语言的特性提供了绝佳的延展性（如 XPath），在数据存储、扩展及高级检索方面具备对 JSON 的优势，而 JSON 则由于比 XML 更加小巧，以及浏览器的内建快速解析支持，使得其更适用于网络数据传输领域。

12.3.3 Browser action 介绍

1. browser action 简介

当安装了一个扩展的时候，有的时候会看到工具条的地址栏右侧多出一个图标，这个就是 browser action 的效果。例如，浏览器上使用了一个 googlemail 的扩展，它的显示如图 12-2 所示，图 12-3 即为 browser action 形成。



图 12-2 工具条的地址栏右侧的 browser action 的效果



图 12-3 googlemail 的扩展图标

作为这个图标的延展，一个 browser action 图标还可以有 tooltip、badge 和 popup 形成。注意：Packaged apps 不能使用 browser actions。再给出另外一个 browser action 的例子，如图 12-4 所示，地址栏右侧的彩色正方形是一个 browser action 的图标，图标下面的是 popup 页面效果。

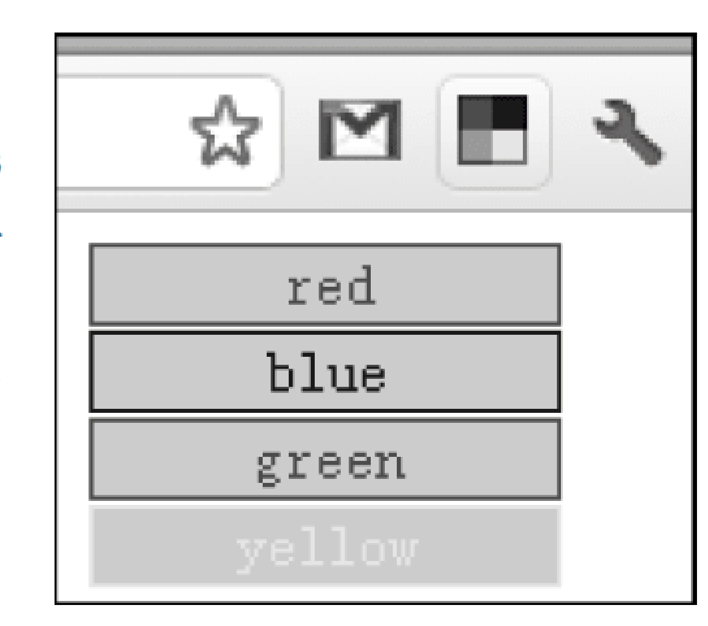


图 12-4 color

browser action 的使用方法：在 extension manifest 中用下面的方式注册 browser action。

```
{ "name": "My extension", ...
"browser_action":
 {  "default_icon": "images/icon1.png",      // 可选
    "default_title": "Google Mail",          // 可选,提示信息
     "default_popup": "popup.html"           // 可选,弹出页面
 }, ...
}
```

在上面的代码中注意逗号的使用,有的时候逗号的缺失也可能导致程序的失败。

2. browser_action UI 的组成部分

前面讲过,一个 browser action 可以拥有一个显示图标、一个 tooltip、一个 badge 和 popup。

1) 图标

browser action 图标推荐使用宽高都为 19 像素,更大的图标会被缩小。静态图片可以是任意 WebKit(上面说了 Chrome 的浏览器引擎为 Webkit)支持的格式,包括 BMP、GIF、ICO、JPEG 或 PNG 格式的图片。

设置方式为:修改 browser_action 的 manifest 中 default_icon 字段。

2) Tooltip

修改 browser_action 的 manifest 中 default_title 字段。

3) Badge

Browser actions 可以选择性地显示一个 badge——在图标上显示一些文本,如图 12-5 所示的 googlemail 图标上的 3 即为 badge 的显示效果。

Badges 可以很简单地为 browser action 更新一些小的扩展状态提示信息。要设置 badge 文字和颜色,可以分别使用 setBadgeText()andsetBadgeBackgroundColor()。

现在对 browser action 已经有了大概的认识,那么在具体的开发过程中要怎么做才能提高用户的体验感呢?

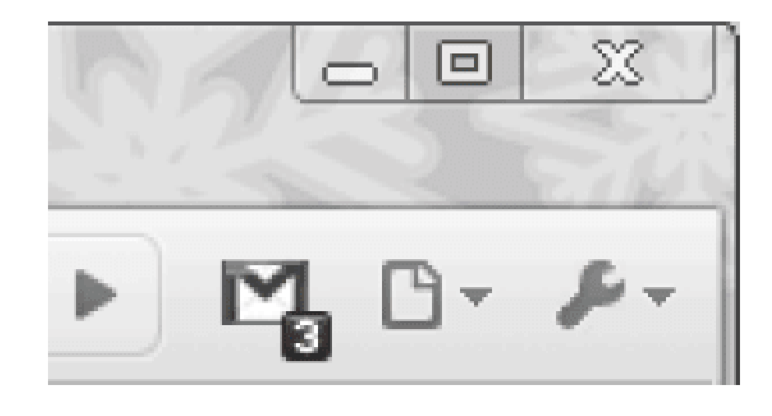


图 12-5 googlemail 图标

为了获得最佳的显示效果,请遵循以下原则:

- 确认 Browser actions 只使用在大多数网站都有功能需求的场景下,确认 Browser actions 没有使用在少数网页才有功能的场景。如果用在少数网页才具有的功能,请参考使用 page actions。
- 确认你的图标尺寸尽量占满 19×19 的像素空间。Browser action 的图标应该看起来比 page action 的图标更大、更重。
- 不要尝试模仿 Google Chrome 的扳手图标,在不同的 themes 下它们的表现可能出现不同的问题,另外扩展应该醒目些。
- 尽量使用 alpha 通道并且柔滑你的图标边缘,因为很多用户使用 themes,你的图标应该在各种背景下都有不错的表现。
- 不要不停闪动插件的图标。

12.3.4 page action 介绍

1. page action 简介

前面多次提到 page action,它到底是什么呢?

如果想创建一个不总是可见的图标，可以使用 page_action 来代替 browser_action。使用 page actions 可把图标放置到地址栏中。page actions 定义需要处理的页面的事件,但是它们不是适用于所有页面的。

注意：打包的应用程序不能使用 page actions。

page action 的使用方法：在 extension manifest 中用下面的方式注册 page action。

```
{ "name": "My extension", ...
"page_action":
{ "default_icon": "icons/foo.png",      // 可选,此处图标的值为相对地址
"default_title": "Do action",           // 可选,显示提示信息
"default_popup": "popup.html"           // 可选 }, ...
}
```

2. page_action UI 的组成部分

同 browser action 一样,page action 可以有图标、提示信息、弹出窗口。但没有 badge,正因如此,作为辅助,page action 可以有显示和消失两种状态。show()和 hide()可以显示和隐藏 page action。默认情况下 page action 是隐藏的。当要显示时,需要指定图标所在的标签页,图标显示后会一直可见,直到该标签页关闭或开始显示不同的 URL(如：用户点击了一个链接)。

与 browser action 相对应的,为了获得最佳的视觉效果,请遵循下列规则：

- 要只对少数页面使用 page action。
- 不要对大多数页面使用它,如果功能需要,那就使用 browser action 代替。
- 图标出现动画,真的会让人很烦,所以还是让它远离 page action。

如果 browser action 拥有一个 popup,popup 会在用户点击图标后出现。popup 可以包含任意你想要的 HTML 内容,并且会自适应大小。如图 12-6 即为 pupup 页面的显示效果。



图 12-6 popup 显示效果

在 browser action 中添加一个 popup,创建弹出的内容的 HTML 文件。修改 browser_action 的 manifest 中 default_popup 字段来指定 HTML 文件,或者调用 setPopup()方法。

3. browser action 与 page action 的区别

(1) popup.html 中定义的 Javascript 变量会在 popup.html 页面关闭时被释放。

(2) background.html 可以保存一些一直需要使用的变量的作用,这里定义的 JavaScript 变量会在 Chrome 浏览器生命期中一直存在,可以把要一直存在的数据保存在这里。

先在 background.html 中定义好变量：

```
var?global_email = "";
```

然后在 popup.html 中用以下方式来引用这些变量：

```
var bgpg = chrome.extension.getBackgroundPage();
bgpg.global_email = "somebody@domain.com";
```

工作原理如图 12-7 所示。

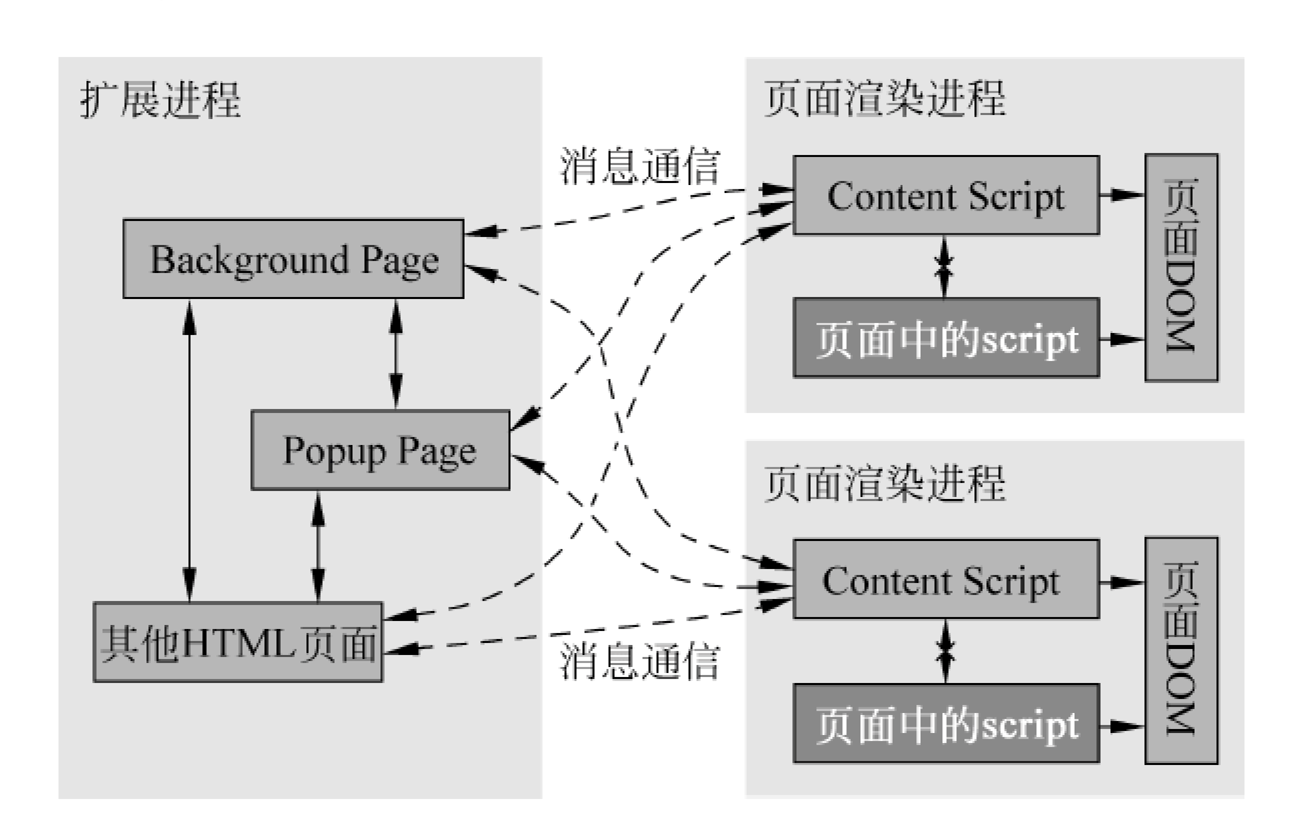


图 12-7 popup page 的工作原理

4. page action 实例：helloworld、sandwish

例 12-1：page action HelloWorld 效果如图 12-8 所示。

项目结构如下：

```
Helloworld/
    manifest.json
    popup.html
```

图标 icon.png(见图 12-9)也放在 Helloworld 的目录下。

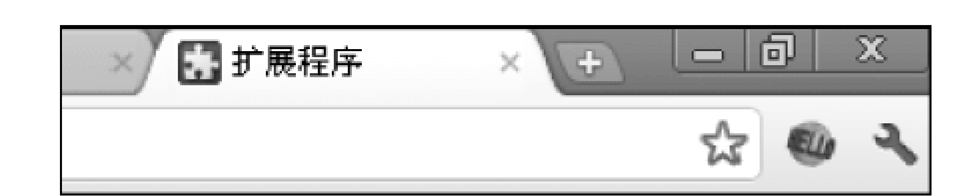


图 12-8 page action 的 HelloWorld 效果图



图 12-9 icon.png

- Manifest.json 文件。

```
{
    "name": "Hello World 插件",
    "version": "1.0",
     "description": "第一个 Chrome 插件",
     "browser_action": {
    "default_icon": "icon.png",
    "popup": "HelloWorld.html"
    }
}
```

- HelloWorld.html 文件。

```
<p>Hello,World!</p>
<p><a href="https://chrome.google.com/extensions" target="_blank">welcome</a>
```

当然，也可以把 href 中的内容换成任何可访问的网址。

例 12-2：显示一个 page action 包含 sandwich 字符串的页面（如图 12-10 所示）。



图 12-10　page action 示例 sandwish

安装这个扩展后，每当你访问的页面地址中有 sandwish 这个单词，就会在地址栏内最右侧出现这个诱人的三明治图标。在此，将给出实现 sandwish 效果的所有代码。

在开发此扩展中需要的文件有 Manifest.json、Background.html 文件和 3 个不同大小的图片。用到的图片如图 12-11～图 12-13 所示。

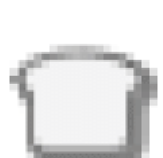
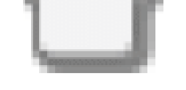



图 12-11　图标 19×19




图 12-12　图标 48×48

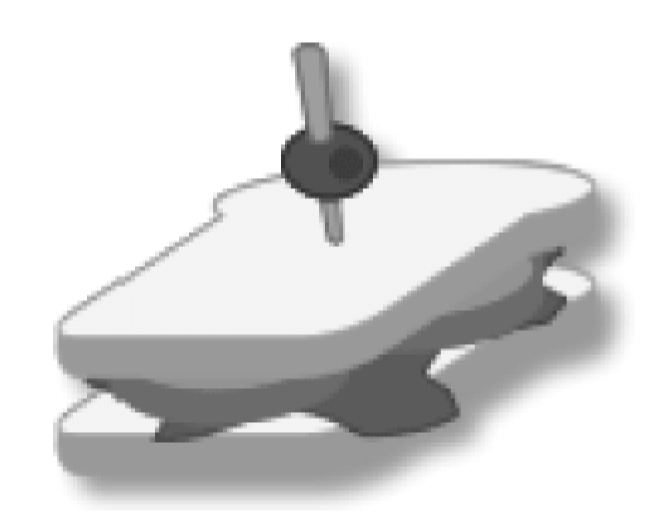


图 12-13　图标 128×128

- Manifest.json 文件。

```
{
  "name" : "Page action by content",
  "version" : "1.0",
  "description" : "Shows a page action for HTML pages containing the word 'sandwich'",
  "background_page" : "background.html",
  "page_action" :
  {
    "default_icon" : "sandwich-19.png",
    "default_title" : "There's a 'sandwich' in this page!"
  },
  "content_scripts" : [
    {
      "matches" : [
        "http://*/*",
        "https://*/*"
      ],
      "js" : ["contentscript.js"],
      "run_at" : "document_idle",
      "all_frames" : false
    }
  ],
```

```
    "icons" : {
      "48" : "sandwich-48.png",
      "128" : "sandwich-128.png"
    }
  }
```

- Background.html 文件。

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      function onRequest(request, sender, sendResponse) {
        chrome.pageAction.show(sender.tab.id);
        sendResponse({});
      };
      chrome.extension.onRequest.addListener(onRequest);
    </script>
  </head>
</html>
```

12.3.5 content script 介绍

1. content script 简介

在上面的例子中看到有 content script 出现，什么是 content script 呢？

content script 是在 Web 页面内运行的 Javascript 脚本。通过使用标准的 DOM，它们可以获取浏览器所访问页面的详细信息，并可以修改这些信息。

1）content script 能做什么

（1）从页面中找到没有写成超链接形式的 URL，并将它们转成超链接。

（2）放大页面字体使文字更清晰。

（3）找到并处理 DOM 中的 microformat。

2）content script 不能做什么

（1）不能使用除了 chrome.extension 之外的 chrome. * 的接口。

（2）不能访问它所在扩展中定义的函数和变量。

（3）不能访问 web 页面或其他 content script 中定义的函数和变量。

（4）不能做 cross-site XMLHttpRequests。

content script 可以使用 messages 机制与它所在的扩展通信，来间接使用 chrome. * 接口或访问扩展数据。content scripts 还可以通过共享的 DOM 来与 Web 页面通信。

2. content scripts 的注册

在 manifest 中添加 content scripts：

```
{ "name": "My extension", ...
"content_scripts": [
```

```
{
 "matches": ["http://www.google.com/ * "],
"css": ["mystyles.css"],
"js": ["jquery.js", "myscript.js"]
 }
],
 ... }
```

使用 content_scripts 字段，一个扩展可以向一个页面注入多个 content_script 脚本；每个 content script 脚本之间用逗号隔开，每个 content script 脚本可以包括多个 Javascript 脚本和 css 文件。现在看一个 content script 的 helloworld 示例，文件结构如下：

```
contentscriptHelloWorld/
    manifest.json
    hello.js
    content.js
```

- manifest.json 文件。

```
{
   "name": "helloworld 插件 2",
  "version": "1.0",
   "content_scripts": [
   {
     "js": [ "hello.js" ],
     "matches": [ "http:// * .google.com.hk/ * "],
     "run_at": "document_end"
   },
   {
     "js": [ "content.js" ],
     "matches": [ "http:// * .baidu.com/ * "],
     "run_at": "document_end"
   }
  ]
}
```

Run_at 属性：如果是 document_end，则文件将在创建完 DOM 之后，但还没有加载类似于图片或 frame 等的子资源前立刻注入。所以在上页效果图中，并没有加载出 google 的 logo 图片。

- hello.js 文件。

```
alert('hello world');
alert(document.title);
```

- content.js 文件。

```
alert('content script');
alert(document.title);
```

查看效果：

（1）在地址栏中输入 http://www.google.com.hk/查看效果如图 12-14 所示（注意 google 的 logo 图片的加载情况）。

（2）在地址栏中输入 http://www.baidu.com/查看效果。



图 12-14 content script 效果图

3. include 和 exclude 语句

一个 content script 被注入页面的条件是：页面 url 匹配任意一条 match 模式，并且匹配任意一条 include glob 模式，并且不匹配任何 exclude glob 模式。由于 matches 属性是必选的，因此 include glob 和 exclude glob 都只能用来限制哪些匹配的页面会被影响。

这两个属性与 matches 属性的语法是不同的，它们更灵活一些。在这两个属性中可以包含“ * ”号和“?”号作为通配符。其中“ * ”可以匹配任意长度的字符串，而“?”匹配任意的单个字符。例如，语句"http://???.example.com/foo/ * "可以匹配下面的所有情况：

```
"http://www.example.com/foo/bar"
"http://the.example.com/foo/"
```

但是它不能匹配下面的这些情况：

```
"http://my.example.com/foo/bar"
"http://example.com/foo/"
"http://www.example.com/foo"
```

4. content script 执行环境

content script 是在一个特殊环境中运行的，这个环境成为 isolated world（隔离环境）。它们可以访问所注入页面的 DOM，但是不能访问里面的任何 Javascript 变量和函数。对每个 content script 来说，就像除了它自己之外再没有其他脚本在运行。反过来也是成立的：页面中的 Javascript 也不能访问 content script 中的任何变量和函数。

现在，将下面这个 contentscript.js 脚本注入 hello.html。

- contentscript.js 脚本。

```
var greeting = "hola, "; var button = document.getElementById("mybutton");
button.person_name = "Roberto"; button.addEventListener("click", function()
{ alert(greeting + button.person_name + "."); }, false);
```

然后，如果按下按钮，可以同时看到两个问候。

隔离环境使得 content script 可以修改它的 Javascript 环境而不必担心会与这个页面上的其他 content script 冲突。例如，一个 content script 可以包含 JQuery v1 而页面可以包含 JQuery v2，它们之间不会产生冲突。

另一个重要的优点是，隔离环境可以将页面上的脚本与扩展中的脚本完全隔离开。这使得开发者可以在 content script 中提供更多的功能，而不让 Web 页面利用它们。

5. 与嵌入的页面通信以及安全性

尽管 content script 的执行环境与所在的页面是隔离的，但它们还是共享了页面的 DOM。如果页面需要与 content script 通信(或者通过 content script 与扩展通信)，就必须通过这个共享的 DOM。

在写 content script 的时候，有两个安全问题必须注意：

(1) 首先，要小心不要给原页面带来新的安全漏洞。

(2) 其次，尽管在独立环境中运行 content script 的机制已经提供了一些保护，如果不加区分地使用 Web 页面上的内容还是可以被恶意的 Web 页面攻击的。

目录结构如图 12-15 所示。



图 12-15 email_this_page 目录结构

其中，email_16×16、mail_128×128 分别如图 12-16 和图 12-17 所示。

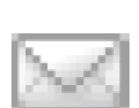


图 12-16 email_16×16



图 12-17 mail_128×128

- 文件 manifest. json。

```
{
  "name": "Email this page (by Google)",
  "description": "This extension adds an email button to the toolbar which allows you to email
the page link using your default mail client or Gmail.",
  "version": "1.2.5",
  "background_page": "background.html",
  "icons": { "128": "mail_128×128.png" },
  "options_page": "options.html",
  "permissions": [
    "tabs", "http://*/*", "https://*/*"
  ],
  "browser_action": {
```

```
    "default_title": "Email this page",
    "default_icon": "email_16×16.png"
  }
}
```

- 文件 background.html。

```
<html>
<head>
<script>
  function customMailtoUrl() {
    if (window.localStorage == null)
      return "";
    if (window.localStorage.customMailtoUrl == null)
      return "";
    return window.localStorage.customMailtoUrl;
  }

  function executeMailto(tab_id, subject, body, selection) {
    var default_handler = customMailtoUrl().length == 0;

    var action_url = "mailto:?"
    if (subject.length > 0)
      action_url += "subject=" + encodeURIComponent(subject) + "&";

    if (body.length > 0) {
      action_url += "body=" + encodeURIComponent(body);


      if (selection.length > 0) {
        action_url += encodeURIComponent("\n\n") +
                      encodeURIComponent(selection);
      }
    }

    if (!default_handler) {

      var custom_url = customMailtoUrl();
      action_url = custom_url.replace("%s", encodeURIComponent(action_url));
      console.log('Custom url: ' + action_url);
      chrome.tabs.create({ url: action_url });
    } else {

      console.log('Action url: ' + action_url);
      chrome.tabs.update(tab_id, { url: action_url });
    }
  }

  chrome.extension.onConnect.addListener(function(port) {
    var tab = port.sender.tab;
    port.onMessage.addListener(function(info) {
      var max_length = 1024;
```

```
      if (info.selection.length > max_length)
        info.selection = info.selection.substring(0, max_length);
      executeMailto(tab.id, info.title, tab.url, info.selection);
    });
  });

  chrome.browserAction.onClicked.addListener(function(tab) {
    if (tab.url.indexOf("http:") != 0 &&
        tab.url.indexOf("https:") != 0) {
      executeMailto(tab.id, "", tab.url, "");
    } else {
      chrome.tabs.executeScript(null, {file: "content_script.js"});
    }
  });
</script>
</head>
</html>
```

• 文件 contentscript.js。

```
var additionalInfo = {
  "title": document.title,
  "selection": window.getSelection().toString()
};

chrome.extension.connect().postMessage(additionalInfo);
```

• 文件 options.html。

```
<html>
<head>
  <title>Options for the Send as Email extension</title>
<style>
#providerSelection {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 10pt;
}
</style>
<script>
  var gmail = "https://mail.google.com/mail/?extsrc=mailto&url=%s";

function toggle(radioButton) {
  if (window.localStorage == null) {
    alert('Local storage is required for changing providers');
    return;
  }

  var selected = radioButton.value;
  if (selected == "gmail") {
    window.localStorage.customMailtoUrl = gmail;
  } else {
    window.localStorage.customMailtoUrl = "";
```

```
    }
  }

  function main() {
    if (window.localStorage == null) {
      alert("LocalStorage must be enabled for changing options.");
      document.getElementById('default').disabled = true;
      document.getElementById('gmail').disabled = true;
      return;
    }

    if (window.localStorage.customMailtoUrl == gmail)
      document.getElementById('gmail').checked = true;
  }
  </script>
  </head>
  <body onload="main()">
  <table border="0">
  <tr>
    <td rowspan="2" valign="top" align="center" width="80">
      <img src="mail_128x128.png" width="64" height="64" />
    </td>
    <td height="22"></td>
  </tr>
  <tr>
    <td valign="center">
      <div id="providerSelection">
      <strong>Select provider to use when emailing a web page address:</strong>
      <br /><br />
      <input id="default" type="radio" name="mailto" value="mailto"
             onclick="toggle(this)" checked>Your default mail handler<br>
      <input id="gmail" type="radio" name="mailto" value="gmail"
             onclick="toggle(this)">Gmail<br>
      </div>
    </td>
  </tr>
  </table>
  </body>
  </html>
```

12.3.6 Theme(主题)

前面在12.3.3节中也多次提到theme,主题是一种特殊的扩展,可以用来改变整个浏览器的外观。主题和标准扩展的打包方式类似,但是主题中不能包含JavaScript或者HTML代码。

Theme在manifest中的字段:colors、images、properties、tints。

1. colors 颜色

采用RGB格式。如果想了解更多颜色值,可以在browser_theme_provider.cc文件“颜色”字段中查找kColor*。

2. images 图片

图片资源使用扩展的根目录作为引用路径。可以通过 browser_theme_provider. cc. 文件中的 kThemeableImages 修改任何图片资源。

3. properties 属性

该字段允许指定主题的属性。例如背景对齐、背景重复和 logo 的轮换。更多的信息可以参考 browser_theme_provider. cc。

4. tints

可以将 tints 应用到按钮、框架 UI、背景标签等用户界面。Google Chrome 支持 tints，但是不支持图片。因为图片不能跨平台工作，并且当添加一些新的按钮时会变得很脆弱。如果想了解 tints，在 browser_theme_provider. cc 文件的 tints 字段中查找 kTint * 。

Tints 采用 Hue-Saturation-Lightness (HSL)格式，浮点数范围为 0～1.0。

Hue 是一个绝对值，只包含 0 和 1。

Saturation 是相对于当前图片的。0.5 表示不变，0 表示完全稀释，1 表示全部饱和。

Lightness 也是一个相对值。0.5 表示不变，0 表示有像素是黑色，1 表示所有像素是白色。

可以选择－1。0 表示任何 HSL 的值都不变。

12.3.7 权限

可能用到的权限清单如表 12-1 所示。

表 12-1 权限清单

权　　限	描　　述
match pattern	指定一台主机的权限。如果需要交互运行网页上的代码。该属性是必需的，具有很多扩展功能，如跨域请求 XMLHttpRequests、注入的内容脚本编程以及 CoookiesAPI 需要主机的权限
"bookmarks"	详见 chrome. bookmarks 模块
"chrome://favicon/"	"chrome://favicon/url" 的形式用于显示页面的 favicon。如：为了显示 http://www. google. com/的 favicon，要声明"chrome://favicon/" 权限代码如下：<img src="chrome://favicon/http://www. google. com/">
"contextMenus"	详见 chrome. contextMenus 模块
"cookies"	详见 chrome. cookies 模块
"experimental"	详见 chrome. experimental. * APIs
"geolocation"	允许 extension 程序使用 HTML5 geolocation API，不需要用户的许可权限
"history"	详见 chrome. history 模块
"idle"	详见 chrome. idle 模块
"notifications"	允许 extension 程序使用 HTML5 notification API，不需要访问权限方法（比如 checkPermission()）。详见 Desktop Notifications
"tabs"	详见 chrome. tabs 或 chrome. windows 模块
"unlimitedStorage"	提供一个用于存储 HTML5 的客户端的数据，如数据库和本地存储的文件，不设限额。如果没有这个权限，扩展限制为 5MB 本地存储空间

12.3.8 消息传递

自从 content script 内容运行在网页环境而不是在扩展中,经常需要一些方法和其余的扩展进行通信。例如,一个 RSS 阅读扩展可能会使用 content scripts 去检测 RSS 阅读扩展应该提供给哪个页面,然后通知后台页面以便显示一个页面交互图标。

通过使用消息传递机制在扩展和 content scripts 中通信,任何一方可以收到另一方传递来的消息,并且在相同的通道上答复。这个消息可以包含任何一个有效的 JSON 对象(null、boolean、number、string、array 或 object)。这里有一些简单的 API 对于 一次简单的请求,还有更多复杂的 API,它们可以帮助你长时间保持连接在一个共享的环境中交换大量的信息,也可以帮助你传递消息给另外一个你知道 ID 的扩展,关于在扩展之间的消息传递本节会有详细的讲解。

1. 一次简单的请求

如果仅仅需要给自己的扩展的另外一部分发送一个消息(可选的是否得到答复),可以使用简单的 chrome. extension. sendRequest()或者 chrome. tabs. sendRequest()方法。这个方法可以帮助你传送一次 JSON 序列化消息从 content script 到扩展,反之亦然。如果接受消息的一方存在,可选的回调参数允许处理传回来的消息。

像下面这个例子一样,可以从 content script 发起一个请求。

- contentscript. js 文件。

```
chrome.extension.sendRequest({greeting: "hello"}, function(response) {
  console.log(response.farewell);
});
```

传递一个请求到扩展很容易,你需要指定哪个标签发起这个请求。下面这个例子展示了如何指定标签发起一个请求。

- background. html 文件。

```
chrome.tabs.getSelected(null, function(tab) {
  chrome.tabs.sendRequest(tab.id, {greeting: "hello"}, function(response) {
    console.log(response.farewell);
  });
});
```

接收消息的一方需要启动一个 chrome. extension. onRequest 事件监听器用来处理消息。这个方法在 content script 和扩展中都是一样的。这个请求将会保留直到做出了回应。下面的这个例子是一个很好的做法——调用一个空对象请求然后得到答复的例子。

```
chrome.extension.onRequest.addListener(
  function(request, sender, sendResponse) {
    console.log(sender.tab ?
                "from a content script:" + sender.tab.url :
                "from the extension");
```

```
    if (request.greeting == "hello")
      sendResponse({farewell: "goodbye"});
    else
      sendResponse({}); // snub them.
  });
```

小提示：如果多个页面都发起了相同的请求，都在等待答复，那么只有第一个发起请求的页面会得到响应，其他的将会被忽略。

2. 长时间地保持连接

有时候持续长时间地保持会话会比一次简单的请求有用。可以建立一个长时间存在的通道从 content script 到扩展，反之亦然，使用 chrome.extension.connect()或者 chrome.tabs.connect()方法，可以对这个通道命名，以区分不同类型的连接。

一个比较有用的例子就是自动填写表单扩展。content script 可以建立一个通道在登录页面和扩展之间，同时发出一条消息给扩展，告诉扩展需要填写的内容。共享的连接允许扩展保持共享状态，从而连接几个来自 content script 的消息。

当建立连接，两端都有一个 Port 对象通过这个连接发送和接收消息。下面展示了如何从 content script 建立一个通道，然后发送和接收消息。

- contentscript.js。

```
var port = chrome.extension.connect({name: "knockknock"});
port.postMessage({joke: "Knock knock"});
port.onMessage.addListener(function(msg) {
  if (msg.question == "Who's there?")
    port.postMessage({answer: "Madame"});
  else if (msg.question == "Madame who?")
    port.postMessage({answer: "Madame... Bovary"});
});
```

从扩展到 content script 发送一个请求看起来非常简单，除了需要指定哪个标签需要连接，简单的办法就是上面例子中的 chrome.tabs.connect(tabId, {name: "knockknock"})。

为了处理正在等待的连接，需要用 chrome.extension.onConnect 事件监听器，对于 content script 或者扩展页面，这个方法都是一样的，但扩展的另外一个部分调用"connect()"，这个事件一旦被触发，通过这个连接可以利用 Port 对象进行发送和接收消息，下面的例子展示了如何处理连接：

```
chrome.extension.onConnect.addListener(function(port) {
  console.assert(port.name == "knockknock");
  port.onMessage.addListener(function(msg) {
    if (msg.joke == "Knock knock")
      port.postMessage({question: "Who's there?"});
    else if (msg.answer == "Madame")
      port.postMessage({question: "Madame who?"});
    else if (msg.answer == "Madame... Bovary")
      port.postMessage({question: "I don't get it."});
```

```
    });
  });
```

可能想知道这个连接什么时候被关闭。例如，如果在为每个开放的端口维护分开的状态，如果想知道它什么时候关闭，就需要监听 Port. onDisconnect 事件，这个事件被激发，要么是通道调用了 Port. disconnect()这个方法，要么这个页面包含的端口没有被加载(例如这个标签是个导航栏)，onDisconnect()保证仅仅被激发一次。

下面介绍一下扩展之间的消息传递，除了在扩展的组件之间传送消息，还可以使用消息 API 来和其他的扩展之间进行通信。既可以使用一个其他扩展，也可以利用的公共 API。

对于扩展内部来说，监听一个传入的请求和连接是一样的，可以使用 chrome. extension. onRequestExternal 或者 chrome. extension. onConnectExternal 方法，如下面的例子所示：

```
// For simple requests:
chrome.extension.onRequestExternal.addListener(
  function(request, sender, sendResponse) {
    if (sender.id == blacklistedExtension)
      sendResponse({});  // don't allow this extension access
    else if (request.getTargetData)
      sendResponse({targetData: targetData});
    else if (request.activateLasers) {
      var success = activateLasers();
      sendResponse({activateLasers: success});
    }
  });
// For long-lived connections:
chrome.extension.onConnectExternal.addListener(function(port) {
  port.onMessage.addListener(function(msg) {
    // See other examples for sample onMessage handlers.
  });
});
```

同样，传递一个消息到另外一个扩展和把消息传递给自己扩展的另外一部分是一样的，唯一不同的是必须知道所要传给消息的扩展的 ID，例如：

```
// The ID of the extension we want to talk to.
var laserExtensionId = "abcdefghijklmnoabcdefhijklmnoabc";
// Make a simple request:
chrome.extension.sendRequest(laserExtensionId, {getTargetData: true},
  function(response) {
    if (targetInRange(response.targetData))
      chrome.extension.sendRequest(laserExtensionId, {activateLasers: true});
  });
// Start a long-running conversation:
var port = chrome.extension.connect(laserExtensionId);
port.postMessage(...);
```

12.3.9 安全策略

无论是从 content script 还是从扩展接收消息，页面不应该是跨站点的脚本，特别是避免使用那些不安全的 API，例如下面的例子：

- Background1.html。

```
chrome.tabs.sendRequest(tab.id, {greeting: "hello"}, function(response) {
  // WARNING! Might be evaluating an evil script!
  var resp = eval("(" + response.farewell + ")");
});
```

- Background2.html。

```
chrome.tabs.sendRequest(tab.id, {greeting: "hello"}, function(response) {
  // WARNING! Might be injecting a malicious script!
  document.getElementById("resp").innerHTML = response.farewell;
});
```

相反地，选择更安全的 API 而不是运行脚本。

- Background3.html。

```
chrome.tabs.sendRequest(tab.id, {greeting: "hello"}, function(response) {
  // JSON.parse does not evaluate the attacker's scripts.
  var resp = JSON.parse(response.farewell);
});
```

- Background4.html。

```
chrome.tabs.sendRequest(tab.id, {greeting: "hello"}, function(response) {
  // innerText does not let the attacker inject HTML elements.
  document.getElementById("resp").innerText = response.farewell;
});
```

12.3.10 APP 打包

前面曾提到，扩展文件是一个签名的 ZIP 文件，扩展名是 crx，比如 myextension.crx。

注意：如果使用 Chrome Developer Dashboard 发布扩展，那么将无须自己打包。自己打包一个 crx 的唯一原因是需要发布一个非公开版本，比如一个 alpha 测试版本给测试用户。

当打包一个扩展时，这个扩展将获得唯一的一对密钥，其中的公共密钥用于标识这个扩展，私密密钥用于保存私密信息和给这个扩展的各个版本签名。为扩展打包的步骤如下：

(1) 访问如下 URL 进入扩展管理页面 chrome://extensions。

(2) 如果开发人员模式边上有"＋"，点击"＋"以展开开发人员模式。

(3) 单击"打包扩展程序"按钮，会出现一个如图 12-18 所示的对话框。

在扩展程序根目录中填入扩展所在的目录，如 c:myext(可以忽略其他项，第一次打包



图 12-18　打包扩展程序对话框

时无须指定私钥)。

(4) 单击“确定”按钮。会生成两个文件：

- a. crx——是一个真正的扩展文件，可以被安装。
- a. pem——是私钥文件。

请妥善保存私钥文件，尽可能将它放在安全的地方。在做以下事情的时候，将需要用到它。

(5) 更新这个扩展。

如果扩展打包成功，使用 Chrome Developer Dashboard 上传这个扩展。

- 更新一个包。

为你的扩展创建一个更新版本的步骤如下：

- 增加 manifest. json 文件中的版本号字段。
- 访问如下 URL，进入扩展管理页面 chrome://extensions。
- 单击“打包扩展程序”按钮，会出现一个对话框。
- 在扩展程序根目录中填入扩展所在的目录，如 c:myext。
- 在私有密钥文件中填入私钥所在的位置，如 c:myext. pem。
- 单击“确定”按钮。

参考文献

Firefox 与 Chrome 的设计理念之比较：http://www.ha97.com/2564.html.